中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者介绍

STEVEN FOOTE

Web开发者，就职于LinkedIn。自学编程，热爱技术，尤其是Web技术，持有杨百翰大学（Brigham Young University，会计专业全美第一名）会计专业学士和硕士学位。在攻读硕士学位期间，他搭建了两个AJAX风格Web应用的所有方面，从视觉设计到服务器和数据库运维，以及其他所有相关内容。

一路编程

Learning to Program

[美] Steven Foote 著　佟达 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

这是一本编程入门书籍，然而，如果以书中所讲内容作为入门标准，估计十有八九的在职程序员都不能算已入门。现代软件开发，已经不仅仅是写出正确的代码这么简单，环境、依赖、构建、版本、测试及文档，每一项都对软件是否成功交付起到至关重要的作用，这些都是每一个程序员在开发软件过程中必备的技能。本书对于上述的每一种技能都做了简洁而精练的介绍，以满足最基本的日常软件开发。换句话说，本书实际上是为现代软件开发的入门设下了最基本的门槛。相信每一位本书的读者，不论是即将进入软件行业，还是已经在软件行业工作多年，都会获得收获。

强烈推荐刚刚或将要成为程序员的人及其朋友们阅读本书。

版权贸易合同登记号　图字：01-2015-6645

图书在版编目（CIP）数据

一路编程 /（美）史蒂夫 • 富特（Steven Foote）著；佟达译. —北京：电子工业出版社，2017.1

书名原文：Learning to Program

ISBN 978-7-121-30478-1

Ⅰ. ①一… Ⅱ. ①史… ②佟… Ⅲ. ①程序设计 Ⅳ. ①TP311

中国版本图书馆 CIP 数据核字(2016)第 287278 号

策划编辑：符隆美

责任编辑：徐津平

印　　刷：三河市双峰印刷装订有限公司

装　　订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编 100036

开　　本：787×980　1/16　印张：18.5　字数：350 千字

版　　次：2017 年 1 月第 1 版

印　　次：2017 年 1 月第 1 次印刷

定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

推荐序

编程魔法的麻瓜入门手册

在看一个涉及旧房改造的电视综艺节目时，我突然意识到一件事：同样是需要大量知识和实践作为基础的专业人士，程序员似乎从来没有得到像建筑师那样来自普罗大众的认可与尊重。究其原因，我想可能部分原因是程序员的工作太缺乏可见性，以至于那些不懂技术的麻瓜们对程序员的印象要么是盲目崇拜，要么是盲目轻视，总之没有那种平等而深入的交流方式。尤其在英语极不普及的中国，天书一般的程序代码更是让麻瓜们望而却步，遑论去理解和交流。像本书译者、我的同事佟达所遭遇的“家人不知道我一天到晚在干些啥”的困扰，对于广大中国程序员而言恐怕是家常便饭。

本着帮助麻瓜进入魔法世界的崇高理想，Steven Foote 写了这么一本精彩的编程入门教材。相比常见的高校编程教材，这本《一路编程》最大的好处是它的“不求甚解”——作者堂而皇之地宣称：你不知道一行程序代码背后那些魔法是怎么发生的吗？无所谓！“你不需要知道那些指令是什么，或者理解它们的工作原理，只要它们能用就行。”这么一来，读这本书入门的麻瓜们可就比它们大部分在学校里上编程课的同侪们更有希望：当那些抱着大部头教材的学生们正在被指针地址等高深的魔法细节吓破胆时，这本书的读者倒是——像已经在干着程序员这份工作的人一样——放心大胆地忽视绝大部分细节，写出一两个能运行的程序，并从中获得继续学习必不

可少的成就感。

其实——出乎麻瓜们的意料——绝大部分的软件开发根本不是什么火箭科学。就像我经常跟朋友打趣说的，我们写这些程序连四则运算都用不全，主要除法不怎么用。相比科学，软件开发其实更像一种工匠手艺（craft）。使一个优秀的程序员区别于平庸程序员的，正如 Neal Ford 在《卓有成效的程序员》中所说，是好的习惯和趁手的工具。Foote 很准确地抓住了这一点。在快速介绍 Javascript 编程之后，他迅速而不失全面地介绍了体面的前端程序员应该用到的全套工具箱：命令行、构建工具、数据库、正则表达式、测试工具、文档……招聘时我们常会对着候选人提交的一个光秃秃的源代码文件挠头，只有经历过这种痛苦的人才会知道，候选人提交的代码作业有 Grunt 构建脚本甚至有单元测试是一件多么令人愉悦的事。

处在当今这个数字化浪潮方兴未艾的时代，能让麻瓜顺利学会编程（或者至少对编程有点概念），其意义超出家庭和谐的层面。中国无比巨大的 IT 需求，与有限的 IT 人才供应之间的矛盾，近年来不仅没有缓解，倒有愈演愈烈之势。另外，众多出身、学历平平的青年人有志于投身 IT 这个大有前途（至少有“钱途”）的行业，却苦于迈不过入门的门槛。如果有更多的学校、企业、培训机构照着 Foote 这个思路，用深入浅出、符合直觉、又贴合行业最佳实践的方式来给这些青年人提供指导，不仅能创造若干就业机会，说大一点更是为“互联网＋”宏观战略提供人才支撑，善莫大焉。

最后，祝各位读者阅读愉快，祝佟达及广大程序员都得到家人更多的理解与支持，祝更多麻瓜走进编程这一神秘的魔法世界。

ThoughtWorks 总监咨询师　熊节

2016 年 11 月

译者序

这是最好的时代，也是最坏的时代

在中国，IT 从业者有数百万人之多，但这其中，称得上会编程的，不会超过十分之一。

我所说的会编程，绝不仅仅是会写代码，而是包括环境搭建、版本管理、构建管理、单元测试、文档编写、团队合作，以及任务拆分在内的综合技能。很多人——包括我自己——一开始都无法理解，一个程序员除了写代码，为什么还需要懂这么多东西。

当年在学校，有幸参加一个学生团体，利用课余时间做点小项目。第一次几个人一起写代码，还有点小激动。然而激动并没有延续很久，过了两天，当大家准备把各自写的代码合到一起时，发现这是个根本无法完成的任务，每个人都有自己的想法，从代码组织方式，到代码风格，甚至连 IDE 都不一样。那是第一次感受到，真正的软件开发，和写 C 语言程序设计的作业完全不一样。

2009 年年底，我到微软亚洲研究院的创新工程中心实习，当时的部门负责人是邹欣，他是《移山之道》和《构建之法》的作者。进组的第一天，我就拿到几张纸，上面写着一些基本的编码规范。分配给我的电脑上环境已经准备好，从 TFS 上签出

项目代码，在项目文件中的编译选项已经配置好，直接运行编译，之后执行脚本就可以将服务跑起来。从新人进组到可以开始工作，不到一天时间，真是让我眼界大开。当然，这才是开始，后面从代码提交，到工作项分配，再到上线部署，每一件事都在刷新着我对软件开发的理解。不过，作为一个小小的实习生，那时的我只想安安静静地写代码，没有仔细思考这些专业工作背后的意义。

我毕业后的第一份工作就职于一家研究所，所在的部门人员能力都很强，但是因为信息相对闭塞，对于现代软件开发方法并不是很了解，所以开发团队的很多做法都比较原始，导致需要花费大量的时间和精力在管理代码版本、修复由于更新代码导致一些已有功能不能用的 bug 等上。我尝试将在微软亚洲研究院学到的那些知识引入到团队中，觉得只要搭建起 TFS，就水到渠成了。可惜现实狠狠给我上了一课，搭建 TFS 其实是所有事情中最简单的，设定编码规范、规范代码提交流程、统一编程环境、编写自动化脚本等，每一件事都非常困难。这时我才意识到，如果一个团队中大部分人都不了解这些现代软件开发的知识，靠一两个人去推动，几乎不可能。

后来，我来到 ThoughtWorks，发现这里每个人都能够熟练使用 Git 管理代码，使用 Gulp、Maven、SBT 等管理构建构成，还会写大量的自动化测试来保证质量。从代码修改到测试环境上线，只需要 5 分钟时间，整个过程不需要人参与，程序员们只要看着屏幕上的流水线走到最后亮起绿灯，就可以安心地做下一个任务。后来有人给这种工作方式起了个很直观的名字，DevOps，中文叫作“开发自运维”。在这里，我了解到了为什么需要敏捷开发，为什么要做持续集成、持续交付，为什么要组建全功能团队。以前对于软件开发的很多疑惑，都慢慢解开了。我常常想，要是我在还没毕业的时候，就知道这些事情，会少走多少弯路啊！

最近几年，参与了几次技术咨询项目，接触到更多软件开发者。很多业界有名的公司，实际上软件开发人员的技能非常不足。完全不理解软件工程任何概念的程序员大有人在，不会使用命令行工具、不知道如何处理代码冲突、从不做单元测试、基本上全靠网上搜索一些代码片段来完成任务，这样的人真心不能算会编程。当然也不乏一些在日常工作中有思考，能够理解软件开发的痛点，但是苦于不知道如何改进的程序员。因为从来没有人告诉他们怎么做才是正确的编程方式。

现在市面上对于每一个流行的技术都有大量的书籍文档做介绍，然而，唯独缺

少一类书，告诉读者如何才能做一名合格的程序员。事实上，我自己以前也一直认为，要想成为一名合格的程序员，需要读很多不同方面的书。直到我看到了本书。

当时接下这本书的翻译，初衷是想要将这本书送给我当时的女朋友，现在的妻子，因为她刚好也是会计，和本书作者在转行做程序员之前的职业一样。我的本意是通过这本书，让我的妻子也可以对编程感兴趣，能够理解我每天对着电脑到底是在干什么。然而当我翻译到第 4 章，介绍 JavaScript 构建工具那部分时，我发现这本书并不是我一开始想的那样，它不只是一本介绍如何写 JavaScript 代码的入门书。在只有两百多页的书中，作者对所有软件开发相关的技能都做了介绍。对于每个编程必备技能，作者仅仅介绍其在日常开发过程中最常用到的一些知识，用 20%的篇幅，把 80%的场景都覆盖到了。不仅如此，因为作者自己从对编程一无所知开始学习，所以在介绍一些相对难理解的概念时，能够设身处地地从初学者角度着想，用直白的语言，将一些概念解释出来。尽管可能从专业人士角度看，不算非常严谨，但对于日常开发工作来说，刚好够用。

对于程序员来说，这是最好的时代，物联网几乎改变了所有行业，甚至已经有人在提“程序员拯救世界”这种说法，作为程序员，我们应感到自豪；然而，这也是最坏的时代，软件开发对程序员的要求越来越高，20 年前也许会用 HTML 设计网页已经非常厉害，但今天你需要会很多不同的技能才能成为一名合格的程序员。如果你希望能够在软件开发这条路上一直走下去，本书可以帮你迈出坚实的一步。

感谢本书的编辑符隆美，在翻译本书期间给我不少鼓励，让我能够坚持译完本书，没有她的帮助，本书不可能完成。

佟达

2016 年 11 月

前言

我为什么写这本书

像大多数伟大的（会计师）故事一样，我的故事开始于一个 Excel 电子表格。那是 2008 年，我正在犹他州普若佛的杨百翰大学学习会计，同时在法学院图书馆做接待员。一天，我的老板问我知不知道如何在 Excel 里把一列名字随机打乱顺序。“当然，”我骗了他，然后去做了每一个说了大话的人都会做的事：谷歌一下。谷歌给我展示了至少三种能够在 Excel 中打乱列表顺序的不同方法。2 分钟后，我就把打乱顺序的列表交给了老板。就在那时，她认定我很擅长计算机，应该去图书馆系统部工作。我不太确定是不是用谷歌检索东西的天赋就等于擅长计算机，但我很感激她这么认为。

我的编程之旅就是从法学院图书馆系统部开始的。第一天，我的老板给了我一本有着 25 年历史的书——《Perl 编程》，并向我展示了我的桌子，坐落在一个没有窗户的房间，上面杂乱地堆放着老旧的计算机、键盘和显示器。临走之前，他告诉我，他正忙着其他事情，但是在他回来之前，这个 Perl 教材会让我一直有事可做。我翻开书，开始阅读，那是我从没有过的感受。

那本书写于 Windows 尚未出现之时，所以它假定读者在用 UNIX 操作系统（我从没听说过 UNIX）。我忽略了这条，继续使用我的 Windows XP 计算机，它刚刚从图书馆计算机实验室退休（在此之前，我从没听说一台计算机对于一个图书馆计算机实验室来说太老了的这种事）。那本书说，给某个地址寄一封贴好邮票的回邮信封，

就会收到一张存着 Perl 的软盘。我决定不邮寄而使用我的谷歌技能，找到下载和安装 Perl 的说明。我以前下载过软件，但我并不知道程序语言也需要下载安装，我有点紧张，怕把我的计算机搞坏，但我发现没法跳过这一步，只能硬着头皮继续。

那本书第 1 章展示了一些示例代码，用来打印 `Hello, World!`，看起来就像这样：

```
print ("Hello, world!");
```

书中让我把这段示例代码保存到一个名为 `hello_world.pl` 的文件中。我之前唯一用过能往计算机里输入文本的软件就是 Word，于是我打开 Word，开始打字。我大概花了 45 秒意识到 Word 不是写代码的地方，然后花了 45 分钟才找到正确的地方。穷尽我所有谷歌技能，都想不出来该搜索什么才能找到写代码的正确方式。我现在进入了一个全新的、不熟悉的世界，我熟悉的那些检索词都没用。最终，我找到了答案——记事本（正如你将在第 1 章“‘Hello, World!’写下你的第一个程序”中会学到的，记事本并不是正确答案，但它确实可以工作），然后我继续艰难前进，只是有一点点沮丧。

最终，我把书中的代码敲进记事本，并保存文件。什么都没发生。但我“很擅长计算机”，而且我知道如果有什么东西不工作，你应该尝试重启。所以我关掉了这个文件，尝试通过双击重新打开它。一个黑底白字的小窗口在屏幕上一闪而过，持续不到 1 秒就消失了，而且记事本也没有打开。那时，我觉得我把计算机给搞坏了，而且我不太相信我的老板不会因为一台计算机坏了而发火，尽管它已经很旧了。然后我的脑中闪过一个想法，就是我的程序事实上已经执行完了。我穿过房间，跑到打印机旁，想看看我是不是已经成功打印了“`Hello world!`”。打印机空闲着，托盘里也没有任何新打印出来的纸张。我试着重启打印机，以防万一。不走运（在很久之后我才知道 print 意味着打印到屏幕，而不是打印到一张纸上），那天下午剩下的时间就在重复着这种令人抓狂的事情。

那天结束时，我感觉我并不擅长计算机；事实上，计算机在用它们的方式对付我，而且它们好像乐在其中。我很想放弃，但我需要工作，而且我不愿言败，我不想让计算机获胜。接下来的几个月里，我只取得缓慢而零星的一些进展。我不断在原地打转，不得不一遍又一遍学习一些相同的东西。唯一可能有能力指导我、回答我问题的人，不是特别忙，就是太有经验以至于帮不上什么忙。就像我在读的那本

有关 Perl 的书（它假设我已经知道怎么用 C 编程，鬼才知道那是什么东西），这些可能的指导老师都认为我懂得比我实际知道的多得多。我不想泄露我什么都不知道，怕丢掉工作（这可能不是个明智的举动）。然而，即使一开始感到沮丧，我也能看出程序是多么强大，而且甚至还挺享受的。最终，知识碎片开始汇集到一起，我慢慢开始理解。

我的编程入门之路经历了许多的坎坷，而且我意识到，每个希望自学编程的人，都可能有类似的经历。有几次，我想要放弃，觉得编程就是给那些计算机科学系的书呆子学生准备的。编程在刚开始学习时确实有些吓人，而有经验的程序员知道那么多东西，以至于让人不敢对他们提问题。但是不管有多难，不管多少次你可能想要用头撞桌子或是把计算机扔到地上，编程最终会带给你惊人的乐趣和回报。当我意识到编程有多么棒时，我放弃了会计学硕士学位和一个在顶级会计师事务所工作的机会，转而从事编程。我从没后悔。我写下这本书，是因为当年刚开始学编程时，就想有这么一本书。

为什么你应该读这本书

计算机就在我们身边，遍布我们生活的方方面面，然而大多数人还并不真正理解它们的工作方式，或者如何让计算机为我们工作。我们都被限制在计算机已经知道怎么做的那些事情上，但是计算机一直以来都意图成为可编程的机器。你可以为你桌上的计算机编程，让它做任何你想要做的事。随着世界越来越依赖于计算机，编程技能对每个人来说都将是必需的，不仅仅对专业软件工程师或开发者来说是如此。这本书会帮助你学习编程，并且乐在其中。

在接下来的几页，你会建立编程基础，为你实现全部编程目标做好准备。不论你想要成为一个专业的软件开发者，还是想要学习如何更高效地和程序员沟通，或者只是对于程序如何工作感到好奇，这本书都非常适合作为帮你达成所愿的第一步。学习编程仍将困难，但却有可能，并且有希望变得有趣，而不是让人沮丧。

你的项目

最好的学习编程的方式就是真正去编程。贯穿本书，你将编写一个完整的 Chrome 浏览器扩展程序。Chrome 浏览器扩展程序是能够增强（或者扩展）Chrome 浏览器功

能的程序。我们即将搭建的这个扩展程序会向用户询问名字和手机号，然后把用户 Facebook 新鲜事里的所有照片，都根据用户的位置（通过电话号码确定）改成小猫小狗。这个扩展程序（我们称之为猫咪书）可能没什么特别的用处，但搭建它的过程会很有趣，而且期间会帮你学到很多重要的编程概念。

目录

1 “Hello, World”写下第一个程序

注

项目：创建一个谷歌浏览器扩展程序

欢迎来到编程世界！我认为学习编程的最好方式就是写代码（如果你没有错过前言，应该已经知道了），所以我们要在本章写一个程序。贯穿本书，我们将搭建一个大项目，这个程序将是这个大项目的第一部分。如果你不在计算机旁边的话，坐到计算机跟前去吧！你该写点代码啦！

在我们开始之前提醒一句：本章我们会涉及一些内容，你现在理解不了。不理解你在做什么（至少对我来说）是计算机编程的重要组成部分。在书的结尾，全部奥秘都将揭晓，相信我。

选择文本编辑器

编程的最重要组成部分之一就是写代码（不过，就像在下面注释中讨论的，编写代码和编程并不完全等价）。当我说代码，我是指写给计算机的指令，用计算机能理解的编程语言写成。代码是用文本编辑器写成的纯文本文件。你的文本编辑器可能是你最重要的工具（就像罗宾汉的弓、亚瑟王的王者之剑，以及苏珊大妈的嗓音）。有很多文本编辑器供你选择，所以请明智选择。

编程与编写代码

编程与编写代码的区别可能比较细微，但非常重要。编写代码只是编程的一部分。事实上，编写代码很可能是编程最简单的部分。编程包括很多任务，比如创建环境，让你的代码可以在其中成功运行；以合理的方式组织代码；测试你的

代码以识别错误；调试代码，找到产生错误的原因；使用别人写的代码和框架；把你的代码打包，以便其他人可以使用它。当你学会如何编程，编写代码会更加有趣。

提示

我学习编程的第一个错误就是编辑器的选择。我曾经只用过微软 Word 来把我的词语和想法输入计算机，所以我尝试用 Word 编程。没过多久我就意识到这不可行。接着，我打开记事本，并用了几个月，直到我发现一个很棒的替代品，我本该一直用它的。记事本并不包含任何一款优秀编辑器该有的任何核心功能。从我的错误中吸取教训，重要的事情再说一遍，不要用记事本编程！

核心功能

有一些文本编辑器，设计的目的就是用来写代码。下一节概述了一些很流行的文本编辑器和他们最重要的特性。所有这些编辑器都有一些核心功能：等宽字体、语法高亮、文本补全，以及可扩展性。

等宽字体

等宽字体是一类字体，他们的每一个字符都占用相同的空间。换句话说，一个 i 和一个 w 一样宽，同时也和一个空格一样宽。最初你可能觉得这样的字符看起来很丑、很奇怪，但过段时间，你就会开始容忍它，欣赏它，然后发现格式化时等宽字体给你的代码带来的美感。

语法高亮

就像英语有名词、动词、形容词等，编程语言也是由不同部分组成（比如变量、保留字和字符串）。一款好的文本编辑器能够区分编程语言的不同部分，一般通过不同的文字颜色来区分。见图 1.1 和 1.2，你甚至不需要理解这段代码是什么意思，就可以看出图 1.2 可读性更好。语法高亮还能够帮你查找代码中的错别字。

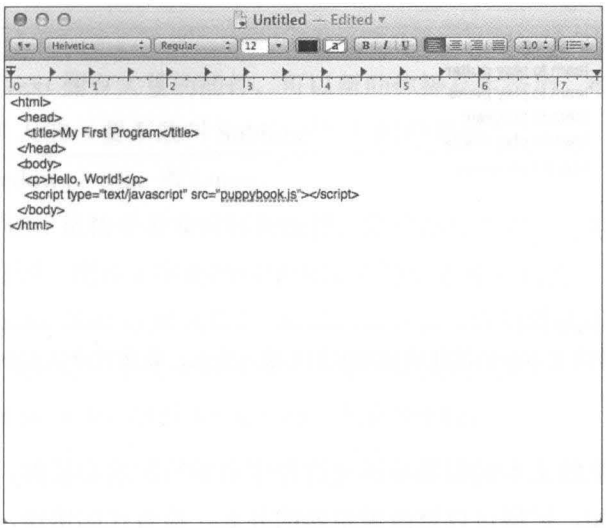

图 1.1 一个非等宽字体格式的 HTML 文档

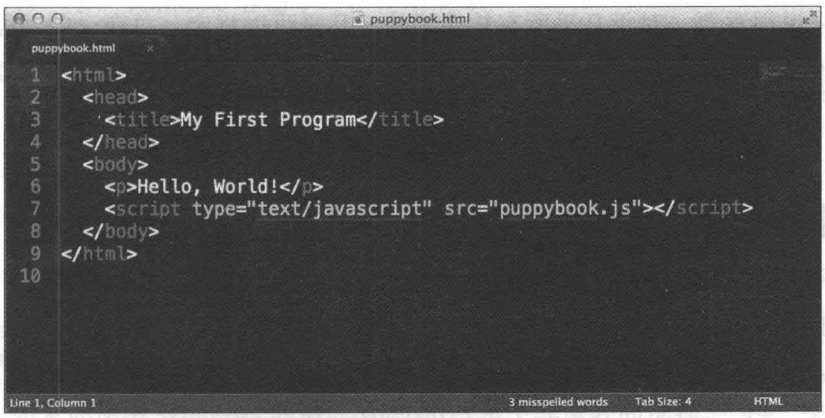

图 1.2 使用等宽字体的同一个 HTML 文档,是不是更容易读（也更漂亮）

文本补全

写代码时，你需要一遍一遍输入同样的单词。对于正常工作的代码，这些单词每次输入必须完全一样（比如，myCode 和 MyCode 或者 mycode 是不一样的）。微小的拼写错误很难查找出来，并可能导致很头疼的问题。每一款优秀的文本编辑器都包括不同程度的文本补全功能——与谷歌能够猜到你想要搜索什么类似，文本编辑器可以猜到你要输入什么（见图 1.3）。使用文本补全能够帮助你更快地写代码，并且还能帮助你避免输入错误和其他错误。

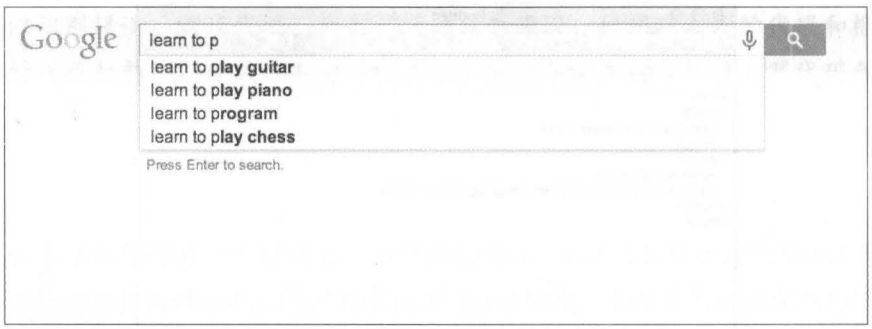

图 1.3　优秀的文本编辑器能够猜测你想要输入什么，就像谷歌猜到你要搜索什么

可扩展性

为程序员而生的文本编辑器应该允许程序员编写扩展和插件，来增强和修改编辑器的行为。比如，扩展可以修改编辑器的外观，检查代码错误，或者给一个文件快速添加“代码片段”。如同你即将在本书中见到的那样，可扩展性不仅仅对于文本编辑器来说很重要，对于几乎所有软件都是。几乎可以肯定，总有一些功能是软件原作者没有想到、没有时间实现或者不想实现的。这些功能可以被其他想要的人实现出来，可扩展性让这成为可能。

做出你的选择

选择对你来说最佳的文本编辑器，依赖于你的个人偏好（和你付费的意愿），以及你要工作的项目类型。我使用 Vim 来做大部分的代码编写工作，但我写 Java 代码时用 IntelliJ，写 iOS 应用时用 Xcode。下面的文本编辑器简介应该能帮你做出选择。

什么是 IDE

在下一章，你会学到计算机到底如何执行你在文本编辑器中写下的代码。这个过程中的一部分包含创建一个能够理解和执行代码的环境。在本书中，我们使用一门叫作 JavaScript 的编程语言；能够理解和执行 JavaScript 的环境是网络浏览器。对于其他一些编程语言，文本编辑器也是理解和执行代码的环境。这类特别的文本编辑器被称为集成开发环境，但程序员们喜欢称他们为 IDE。对于需要 IDE 的语言来说，IDE 是强大的工具。网络浏览器对于写 JavaScript 代码来说可以做到同样强大。

Sublime Text

- Sublime Text 拥有大量的插件，可以帮助你提高生产力。很容易使用，并且具有很吸引人的（要不敢叫 Sublime[1]？）用户界面。
- 支持 Windows、Mac 和 Linux。
- Sublime Text 是初学者非常好的选择。简单而且直观，它遵从了很多文字处理程序的习惯，用过文字处理程序的读者朋友会非常熟悉。
- 一份 Sublime Text 许可证要 70 美元，不过你可以免费试用。
- Sublime Text 是基本文本编辑器，不是全功能的 IDE。
- 可以从 http://www.sublimetext.com/ 下载 Sublime。

TextMate

- TextMate 和 Sublime Text 有很多一样的插件——支持这个的插件通常也支持另一个，TextMate 容易学习，使用简单。
- 只能在 Mac 上使用。
- 和 Sublime Text 一样，TextMate 也是初学者的好选择。你不需要学习任何新东西，就能适用它，并具备高效生产力。
- 一份 TextMate 许可证要 55 美元，不过你可以免费试用。
- TextMate 是基本文本编辑器，不是全功能的 IDE。
- 可以从 http://macromates.com/ 下载 TextMate。

Notepad++

- 如果你使用 Windows 的话，Notepad++是个非常好的选择。“++”来自编程语言 C++；意思是 C++和 C 很像，但更好——所以 Notepad++和 Notepad 很像，但更好。
- 只能用于 Windows。
- Notepad++易于使用，很适合初学者。
- 免费。
- Notepad++是基本文本编辑器，不是全功能的 IDE。

---

1 sublime: adj. 崇高的；壮丽的；宏伟的；令人赞叹的

- 可以从 http://notepad-plus-plus.org/download/ 下载 Notepad++。

Gedit

- Gedit 是一款不错的基础编辑器，不过外观不像其他编辑器那么好看，如果你用 Linux，很可能系统已经默认安装。
- 支持 Windows、Mac 和 Linux。
- Gedit 易于使用，也易于学习。
- 免费。
- Gedit 是基本文本编辑器，不是全功能的 IDE。
- 可以从 https://wiki.gnome.org/Apps/Gedit/ 下载 Gedit。

Vim

- Vim 需要一段时间来学习，不过一旦了解了是在做什么，通过很多快捷键和插件，你可以工作得非常快速。另外，Vim 和几乎每一种操作系统兼容，而且常常预装其中，所以工作在陌生的操作系统上，使用 Vim 会让你感觉不那么陌生。我曾经有些怕 Vim，但现在，它是我最喜欢的编辑器。
- 几乎可以用在已知的每一种操作系统中（已经预装在 Mac OS X 和 Linux 中，但在 Windows 上你得自己安装）。Vim 就像“小强”——几乎能够在任何环境生存下来，即使我们不在了，它还将存活很久。
- Vim 学起来不容易，可能对于初学者来说不是很适合。不过，如果你想要学习 Vim，你可以找到大量的资料，包括程序内置的教程（在命令行中输入 **`vimtutor`**）。
- 免费。
- Vim 是基本文本编辑器，不是全功能的 IDE。
- 如果没有预装 Vim 的话，可以从 www.vim.org/download.php 下载。

Eclipse

- Eclipse 是全功能的 IDE，广泛用于 Java 编程。如果你要在 Java 项目上工作，这是个很好的选择。
- 支持 Windows、Mac 和 Linux。

- Eclipse 对于初学者来说不是特别容易上手，因为它比基本文本编辑器包含的功能多得多。然而，学习 Java 编程，使用 Eclipse 或者 IntelliJ，要比使用上面列出来的编辑器简单得多。
- 免费。
- Eclipse 是一个专注于 Java 开发的 IDE。
- 可以从 www.eclipse.org/downloads/ 下载 Eclipse。

IntelliJ

- IntelliJ 是一个用于 Java 编程的全功能的 IDE，比 Eclipse 更轻量级一点（而且更好看）。IntelliJ 还提供其他语言支持，如 Scala 和 JavaScript。
- 支持 Windows、Mac 和 Linux。
- IntelliJ 学起来和 Eclipse 一样容易。
- IntelliJ 提供免费社区版。一份旗舰版许可证要 199 美元。
- IntelliJ 是一个全功能的 IDE。
- 可以从 www.jetbrains.com/idea/ 下载 IntelliJ。

Xcode

- 如果你要写 iOS 或者 Mac OS X 应用，Xcode 就是给你用的 IDE。Xcode 是苹果公司出的 IDE，其目的在于为苹果平台开发软件。
- 只能用于 Mac。
- Xcode 不容易学习，不过苹果提供了大量文档，而且你可以找到很多社区支持。如果你想要学习开发 iPhone 应用，你需要学习使用 Xcode。
- 免费。
- 可以从 https://developer.apple.com/xcode/ 或 Mac 应用商店下载 Xcode。

Visual Studio

- Visual Studio 是全功能 IDE，主要用于.NET 开发（C#、Visual Basic 等），但也可以用于其他语言。
- 只能用在 Windows 上。
- Visual Studio 学起来和其他全功能 IDE 一样简单——没那么简单。

- Visual Studio 提供了免费的入门版，好用而且实用。付费版 Visual Studio 的价格范围从 1200 美元到 13300 美元。
- 可以从 www.visualstudio.com/downloads/download-visual-studio-vs 下载或购买 Visual Studio。

这里没有列出所有编辑器，但这个清单足够帮你入门。从本书的角度来讲，我推荐 Sublime Text（除非你有使用其他几种编辑器的经验）。Sublime Text 对于我们的目标来说绰绰有余，而且也很容易上手。

创建项目目录

在开始写代码之前，我们需要给代码找个地方存放。计算机程序通常是一组文件一起工作，所以把你需要用到的全部文件组织在一起，放到一个文件夹（文件夹经常被称为目录——本书中我会换着使用文件夹和目录）中，是个不错的主意。我有一个叫作 `projects` 的目录，存放所有这样的项目目录。在你的硬盘驱动器上，创建一个名为 `kittenbook` 的目录。你可以通过文件浏览器（Mac 上叫作 Finder，Windows 上叫作资源管理器，Linux 上叫作 Nautilus）新建目录，点击文件→新建文件夹即可。

从小处着手：创建测试文件

有了项目目录，我们开始编程。你要做的第一件事，是在 `kittenbook` 目录中，创建一个 HTML 文件，命名为 `kittenbook.html`。你可以通过几种不同方式新建文件，但最简单的方式是打开你选择的文本编辑器，创建一个文件（点击文件→新建文件），然后另存为 `kittenbook.html`（文件→另存为，然后找到 `kittenbook` 目录，输入 `kittenbook.html` 作为文件名，并点击保存）。现在，在 **`kittenbook.html`** 文件中添加清单 1.1 中的代码。

清单 1.1　`kittenbook.html`

```
<html>
  <head>
    <title>My First Program</title>
  </head>
  <body>
```

```
    <p>Hello, World!</p>
  </body>
</html>
```

保存文件，然后在文件浏览器中双击它，用网络浏览器打开。你应该会看到类似图 1.4 的页面。

休息一下，享受这一时刻。现在回头看看你的代码，和你在网络浏览器中看到的对比一下。你可以在窗口中看到 `Hello, World!`，你应该在标签页那里看到 `My First Program`。HTML（Hypertext Markup Language，超文本标记语言）由元素组成，一个元素包含起始标签，可选的正文和闭合标签。例如， 是一个起始标签，表示段落元素，`Hello, World` 是正文， 是闭合标签。一个元素的正文还可以包含其他元素，例如清单 1.2。

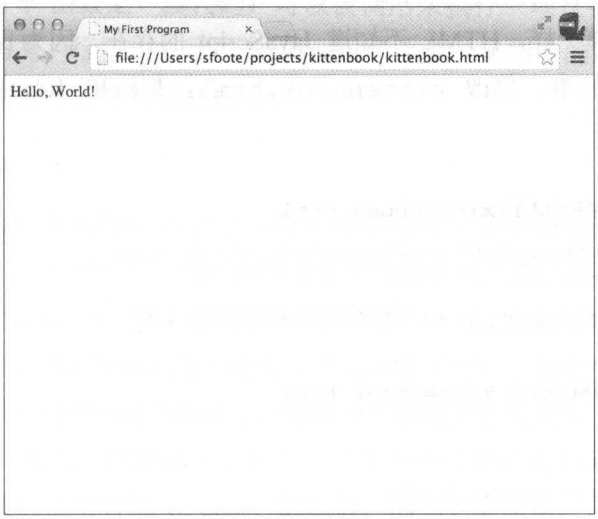

图 1.4 你告诉计算机做的事，成功了！

清单 1.2 `kittenbook.html` 的<head>标签

```
<head>
  <title>My First Program</title>
</head>
```

head 元素包含一个标题元素。这样就告诉了网络浏览器这个 HTML 页面的标题，而大多数网络浏览器会在标签中显示这个标题。你还有很多关于 HTML 的知识要学，但我们暂时先不展开。

HTML 和 JavaScript 如何在浏览器中一起工作

你刚刚创建了第一个网页，现在我们来让这个页面多一点交互。在 kittenbook 目录中新建另一个文件，命名为 kittenbook.js。这是一个 JavaScript 文件，它能让你的网页更有意思。

用文本编辑器打开 kittenbook.js，把清单 1.3 中的代码写到文件中。

清单 1.3 Hello, Friend!

```
alert('Hello, [your name]!');
```

把 [your name] 替换成你的名字。对我来说，就是 alert('Hello, Steven!')。

现在再次打开你的网页（或者刷新它），看看发生了什么。骗你的！页面应该看起来和之前的一模一样。HTML 不知道 JavaScript 的存在，因为我们没告诉 HTML 关于 JavaScript 的事。修改 kittenbook.html，把清单 1.4 中这行代码添加到 <body>元素中。

清单 1.4 添加 JavaScript 到 kittenbook.html

```
<script type="text/javascript" src="kittenbook.js"></script>
```

现在，kittenbook.html 应该看起来像清单 1.5。

清单 1.5 带有 JavaScript 的 kittenbook.html

```
<html>
   <head>
      <title>My First Program</title>
   </head>
   <body>
      <p>Hello, World!</p>
      <script type="text/javascript" src="kittenbook.js"></script>
   </body>
</html>
```

这次，你应该看到类似图 1.5 的结果。

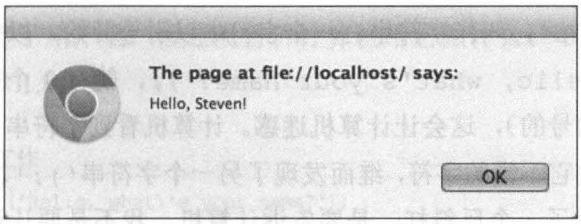

图 1.5 嘿，你写了个真正的程序

太棒了！你让一个窗口上出现了你的名字。你可以输入任何你想要的内容，然后它就会出现在那个窗口上。我们先来说说你刚刚做的事，然后再进入下一个环节。

你写的 JavaScript，是对计算机的一条指令。`alert` 是一种叫作函数的东西，它告诉网络浏览器打开一个带有 OK 按钮的窗口。通过给 `alert` 加上参数，你可以给这个窗口添加文字。函数是一段代码，可以被调用来执行一项任务。参数可以改变函数执行任务的方式。行尾的分号在 JavaScript 中表明一条指令的结束。程序通常有很多指令，所以用分号来分割这些指令。

小幅修改的意义

现在我们要改进我们的程序，让它可以对任何人问好，而不只会说“Hello, World!”。在你改进的过程中，小幅修改，然后在你做更多的修改前，测试那些小改动。最初，这一过程会看起来比较单调和多余。然而，如果你在测试之前做了很多修改，然后发现你的程序不工作，你就很难迅速判断哪个改动搞坏了程序。如果你在测试前只做了一个小改动，然后发现你的程序坏掉了，你就马上知道是什么搞坏了它。

我们第一个小改动，是询问一个名字。一个个性化的问候如果没有名字可不太好。在本例中，我们使用名为 `prompt` 的函数，因为它很简单，而且满足需要。不过，我希望你不要在真实网站中使用 `prompt`（或 `alert`）。尽管 `prompt` 和 `alert` 对于测试和学习很方便，但它们的用户体验很差，而且过时。我们会在第 8 章“函数和方法”中介绍其他替代方法。现在，在 `kittenbook.js` 中，先用 `prompt` 替换 `alert`，把消息修改成“ Hello, what’s your name? ”，见清单 1.6。

清单 1.6 名字提示框

```
prompt('Hello, what\'s your name?');
```

你可能会奇怪，为什么这段代码中是 `what\'s` 而不是 `what's`。一组字符，

像'Hello, World'，叫作字符串。一个字符串以引号开始，以引号结尾。如果我输入 prompt('Hello, what's your name?');，就有 3 个引号（尽管其中一个本来是要当作撇号的），这会让计算机迷惑。计算机看到字符串'Hello, what'，然后发现一些其他它不懂的字符，继而发现了另一个字符串');'。在 what\'s 中，我在单引号前加上了一个反斜杠，是要告诉计算机，我不是要让这个撇号作为字符串结束的标记。这个反斜杠称为转义字符。

现在如果刷新页面，你会看到一个窗口，上面有个地方，让你输入名字，见图 1.6。

注意，当你输入你的名字并点击 OK 时，页面还是会说 Hello, World!。怎么回事？如果你不显示名字，干嘛要我输入它？好吧，我们还没告诉计算机要拿名字做什么。我们做了一个小改动：把 alert 窗口换成了 prompt 窗口。成功了，那么现在我们可以告诉计算机要拿名字做什么。

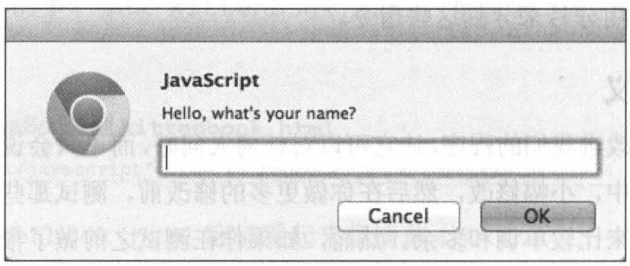

图 1.6　你叫什么名字？

编程中最重要的工具之一就是变量。变量是一个存放数据的地方，这些数据会影响程序的工作方式，第 5 章“数据（类型）、数据（结构）、数据（库）”会详细讨论变量。在这里，我们想要存储从 prompt 窗口得到的名字，见清单 1.7。

清单 1.7　第一个变量

```
var userName = prompt('Hello, what\'s your name?');
```

所有变量都有名字，我们的变量名叫作 userName。我们使用保留字：var 来告诉计算机，我们要创建一个变量。保留字，就是对于编程语言有特殊含义的字，它被保留，所以你不能用这个字作为变量名，否则会迷惑计算机。

现在我们存储了变量，刷新页面，看到一切正常。页面应该看起来和上次刷新

一模一样。我们保存了用户名，但还没告诉计算机用它做什么。下一步，是将用户名插入网页，见清单 1.8。

清单 1.8 让你的变量工作

```
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = 'Hello, ' + userName + '!';
```

我们添加了新的指令，改变了 HTML 文档 document 的 body 部分。还记得 kittenbook.html 的<body>元素么？document.body.innerHTML 指向包含在<body>的起始标签和闭合标签之间的全部内容。document.body.innerHTML = 'Hello, ' + userName + '!';成功地把 HTML 修改成类似清单 1.9 的样子（当提示框让我输入名字时，假设我输入了 Steven）。

清单 1.9 你的 JavaScript 运行过后的 kittenbook.html

```
<html>
  <head>
    <title>My First Program</title>
  </head>
  <body>
    Hello, Steven!
  </body>
</html>
```

恭喜你！你刚刚创建了一个真正的程序。诚然，这不是至今写出来最有用的程序，但它肯定不是最没用的——而且你在过程中学到了很多。

乘胜追击

现在我们把你刚刚开发的程序变成谷歌浏览器扩展。谷歌浏览器扩展是可以安装在谷歌浏览器里的小程序，可以增强用户使用体验。浏览器扩展可以非常强大，有些公司的主要产品就是浏览器扩展。我们本章开始开发的扩展是本书其他章节项目的基础。

第一步，新建一个文件，命名为 manifest.json。这个文件告诉谷歌浏览器关于扩展的信息，以及扩展如何工作。JSON 是一种文档类型，以一种高效的方式保存数据，计算机和人都可以很轻松地读取。在 manifest.json 文件中写入清单 1.10

的内容（记住，每个字符都很重要，包括第一行和最后一行中看起来很奇怪的花括号）。

清单 1.10　谷歌浏览器扩展 manifest.json 示例

```
{
  "manifest_version": 2,
  "name": "kittenbook",
  "description": "Replace photos on Facebook with kittens",
  "version": "0.0.1"
}
```

现在有了 `manifest.json` 文件，你就有了创建 Chrome 扩展程序所需的全部信息。目前，你的扩展程序不会做任何事情，但你还是可以把它加载到 Chrome 浏览器中，以确保 `manifest.json` 文件没有任何问题。要把扩展程序加载到 Chrome 浏览器中，你需要打开 Chrome 浏览器（如果你还没有，需要先下载一个），然后在地址栏中输入 **`chrome://extensions`**。你应该能看到一个类似图 1.7 的页面。

选中“开发者模式”旁边的复选框（因为你现在是开发者啦！）以添加你的扩展程序。现在你应该看到一个按钮，写着“加载正在开发的扩展程序”，见图 1.8。

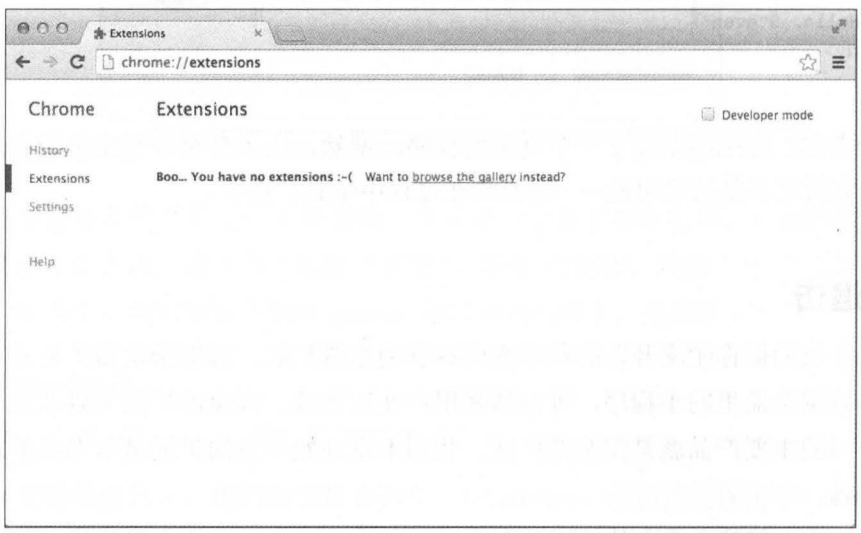

图 1.7　Chrome 扩展程序页面

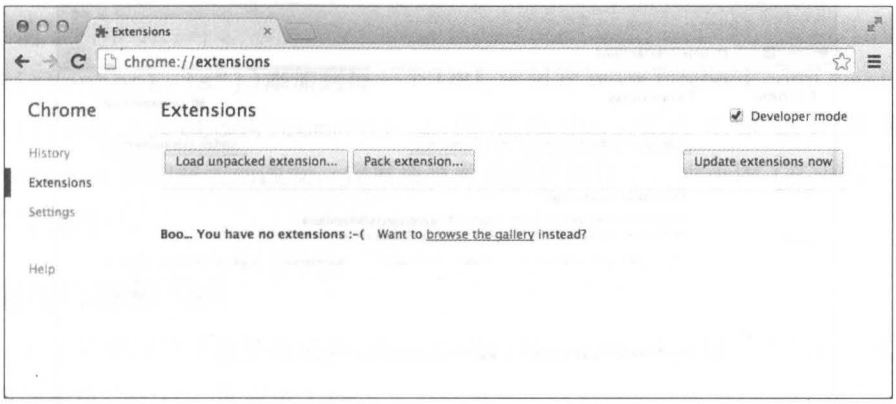

图 1.8 开发者模式的 Chrome 扩展程序页面

单击那个按钮，选择你的整个项目目录（而不是项目目录里的单个文件）。如果选择了正确的目录，并且 `manifest.json` 文件没问题，你就可以看到你的扩展程序加入到了扩展程序列表中，见图 1.9。

如果 `manifest.json` 文件有问题，你会得到一个类似图 1.10 的消息。如果你看到这样的消息，你应该把 `manifest.json` 文件的内容复制并粘贴到一个 JSON 校验器中，比如 http://jsonlint.com/，它会告诉你哪里有错误以及如何修复。

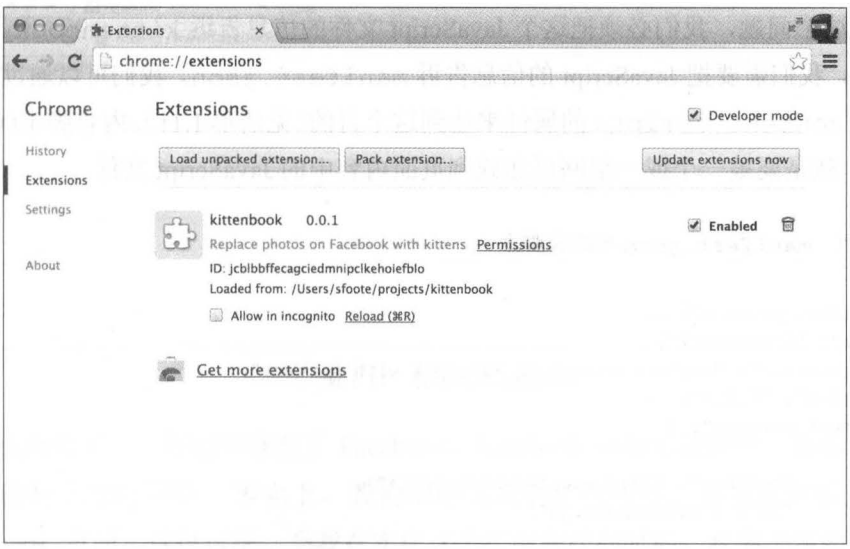

图 1.9 Kittenbook 在 Chrome 中的首次亮相

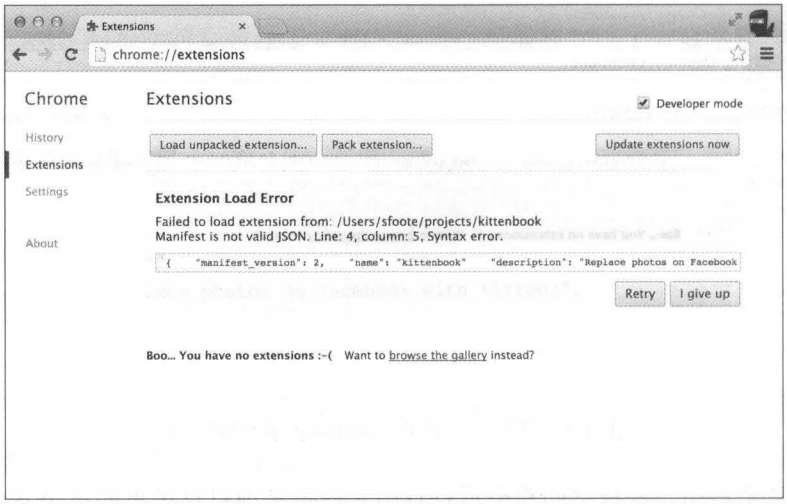

图 1.10　manifest.json 坏了

在 manifest.json 中引用 JavaScript

在 kittenbook 扩展程序成功安装后，你已经准备就绪，要让它真正做点事了。你已经写了一个 JavaScript 文件，你只需让那段 JavaScript 代码在 Facebook.com 上可用。之前我们尝试添加 JavaScript 文件到 kittenbook.html 时，遇到过类似的问题。要解决这个问题，我们必须把这个 JavaScript 文件的信息告诉 kittenbook.html。在这里，我们需要把 JavaScript 的信息告诉 manifest.json。我们可以通过使用一个名为 content_scripts 的属性来达到这个目的（见清单 1.11）。内容脚本（Content Script）就是要被一个或一组网页加载到页面内容中的 JavaScript 文件。

清单 1.11　manifest.json 与内容脚本

```
{
   "manifest_version": 2,
   "name": "kittenbook",
   "description": "Replace photos on Facebook with kittens",
   "version": "0.0.1",
   "content_scripts": [
      {
            "matches": ["*://www.facebook.com/*"],
            "js": ["kittenbook.js"]
      }
   ]
}
```

有了清单 1.11 中新添加的部分，我们的扩展程序就把 `kittenbook.js("js": ["kittenbook.js"])`添加到每一个 URL 中包含 www.facebook.com`("matches": ["*://www.facebook.com/*"] )` 的页面中。要让扩展程序根据你对 manifest.json 做的修改而改变，你需要重新加载扩展程序，点击图 1.9 中的“重新加载”连接即可。

让它运行起来

如果扩展程序重新加载成功，你可以最后对 kittenbook.js 做一点修改，让这句问候看起来更好一点，见清单 1.12。

清单 1.12

```
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = '<h1>Hello, ' + userName + '!</h1>';
```

添加`<h1>`标签，把这句问候设定成标题模式（大号粗体）。现在重新加载扩展程序，打开 Facebook 页面，看看会发生什么。你应该会看到，弹出窗口被打开，类似图 1.11。

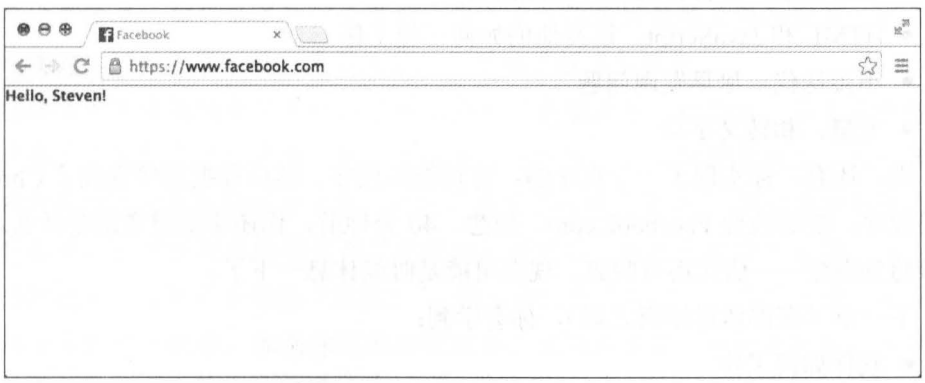

图 1.11 Hello, Facebook!

太神奇了——你刚刚修改了 Facebook。Facebook 在跟你打招呼。看起来很酷，不过没什么实际用处。事实上，我们刚刚实现的扩展程序，会阻止你看到真正的 Facebook 页面，这挺讨厌。当我在实现这个扩展程序的时候，我妻子想要在我电脑上查看 Facebook，她对用一句问候替换掉她的 Facebook 不以为然。当你不是在编写

扩展程序的时候，你可以禁用它，勾选“启用（Enalbed）”复选框即可，见图 1.9。

能力越大，责任越大

读完本章，你已经开始获得非凡的能力。你开始了解，你能够向计算机发送指令，你说什么，计算机就做什么。你不再只是买软件，而是开始做软件了。

本章我们开始做的 Chrome 扩展程序，是一个非常好的工具，帮你了解其他很多编程概念。不过，在目前的状态，这个扩展程序讨人厌，不太合适把它分享给朋友们（特别是如果你没告诉他们怎么禁用它）。你掌握了创造有用、有帮助和有趣的程序的能力，但你也同时掌握了创造讨人厌、有害和恶意的程序的能力。要想着用你的能力做善事。

总结

本章你学到了：

- 文本编辑器，计算机程序被创造出来的地方
- 项目目录，保持你的程序条理清晰
- HTML 和 JavaScript，以及他们如何一起工作
- 小步迭代，尽早发现问题
- 变量，和转义字符

哦，还有，你实现了一个真正的、有功能的程序。然后你把程序变成了 Chrome 扩展程序，能够改变 Facebook.com。想想，40 分钟前，你还不知道变量是什么。你应该感到自豪——估计还有些累。现在可能是时候休息一下了。

下一章（在你休息结束之后），你会学到：

- 软件如何工作
- 软件编译
- 软件解释
- 输入和输出
- 内存和变量

2 软件如何工作

注

项目：合理建立目录结构，拆分 JavaScript 到不同文件中。

在上一章，你已经构建了一个真正的应用软件，但我们隐藏了那个软件（或任何其他软件）如何工作的细节。编程的一个重要部分，就是弄清楚你的程序为什么不工作。如果你不理解软件的工作原理，你就无法弄清楚你的程序为什么不工作。理解了软件如何工作，在思考如何构建程序时，也会让你做出更明智的选择。

我要提醒你一句：这一章会很难，甚至有些痛苦。学习编程很困难，而这一章的内容尤其困难。你可能无法完全理解，没关系。你可以继续后面的章节，或者，如果你愿意，可以再读一遍。重要的是，你开始了解这些概念。

在读本章（或其他章节）时，你可能会有“我要放弃”的想法，我完全理解。编程很难，你可能会因此头痛。我刚开始学习的时候，好几次想要放弃。我还曾经好几次想要把我的电脑扔到地上去（现在还偶尔有这样的想法）。但每当我最终让软件工作起来，就觉得所有的痛苦都是值得的。我问过一些专业软件工程师，他们是否曾想过放弃，答案几乎都是“是的！”，随后他们就开始回忆，那些因为忘记分号或者逗号而抓狂的故事。如果你感觉想要放弃，说明你是在一家好公司里；如果你没有放弃，那证明你是在一家更好的公司里。好好待着吧！

什么是“软件”

从消费者角度看，软件是一个商品，你可以买来运行在电脑或智能手机上。然而你不再只是一个消费者，你已经写了一个程序，所以你现在是一个程序员。本书中，我们要用程序员的角度来看事情。从我们的角度来看，软件就是计算机负责执

行的一系列指令。一个程序可能只是一个包含少量指令的文件，也可能有几百万条指令，分布在几千个相关联的文件中。

从另一个角度看，计算机功能中任何非硬件（即计算机的物理组件）的部分，都可以称为软件。硬件和软件紧密相关：两者缺一不可。没有软件，硬件不知道该做什么，而软件自己无法运行。

第一个软件

虽然硬件和软件缺一不可，但它们的出现总有先后。在这个“先有鸡还是先有蛋”的问题中，我们知道哪个先出现，答案有些出乎意料。1842 年，埃达·洛夫莱斯（Ada Lovelace）写下计算伯努利数的软件，她的软件本来是要运行在查尔斯·巴贝奇（Charles Babbage）的分析机（Analytical Engine ）上，那是一台机械式计算机。但是在 1942 年，分析机还不存在，事实上，分析机一直都没有建成，所以埃达的代码（世界上第一个计算机程序）从来没有被测试过。

软件生命周期

软件在计算机中运行过程可以分为三个基本步骤，见图 2.1。

1．计算机接收一组指令。

2．如果那些指令不是计算机可以理解的形式（二进制），计算机必须尝试（使用其他已经是二进制形式的指令）把它们转换成二进制。

3．计算机硬件执行二进制指令。

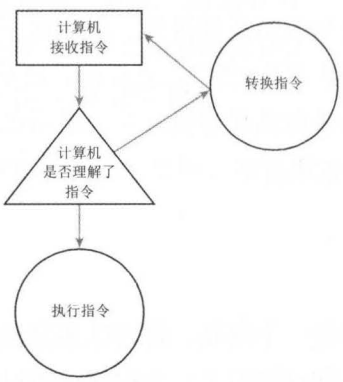

图 2.1　软件如何让计算机做事

源代码——一切开始的地方

在 kittenbook 项目中，你写过源代码。具体来说，你在 kittenbook.js 中写的指令，是这个项目的源代码。其他文件(manifest.json 和 kittenbook.html)也可以当作源代码，但它们并不包含真正的指令。manifest.json 让谷歌浏览器明白你的程序如何工作（输入），而 kittenbook.html 描述了你的网页上应该有什么，以及什么样的格式（输出）。

一组指令

每个程序都是一组指令。有时这些指令在一个文件里，有时散布在不同的文件，甚至不同的计算机中。每条指令单独看都很简单，以 kittenbook.js 为例，你可以看到 3 条指令，见清单 2.1。

清单 2.1 kittenbook.js 指令

```
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = '<h1>Hello, ' + userName + '!</h1>';
```

从这两行代码中，我们可以拆分出 3 条不同的指令。

1. 要求用户输入名字。
2. 告诉计算机，把这个名字保存为一个名为 userName 的变量。
3. 把 document.body 的内容换成字符串'<h1>Hello'+userName+'!</h1>'。

似乎挺有道理，但这些指令里，哪一条告诉计算机显示一个窗口，上面有谷歌浏览器的图标、你输出的消息、文本输入框和两个按钮？哪一条指令告诉计算机把名字存到什么地方？哪一条指令告诉计算机，在 document.body.innerHTML 被赋予一个新值时，该如何更新网页？如果你没有告诉计算机做这些事情，那么是谁？这个魔术是怎么发生的？

上面这些问题的答案是一样的：无所谓。我曾经无法接受这个答案。我刚开始编程时，想要控制一切，还想知道所有东西是如何工作的。事实上，这 3 条指令会被扩展成多得多的指令，你不需要知道那些指令是什么，或者理解他们的工作原理，只要它们能用就行。

你是在一个框架上构建软件。本例中，这个框架就是网络浏览器，prompt 是对

这个框架的一次调用，然后这个框架执行它自己的指令，显示那个窗口。这并不是魔术，因为某个人得把这些指令写出来。但对于你，它就是魔术。

使用他人的工作成果，而不需要完全理解它，这一概念有很多名字，包括抽象层次和黑盒。prompt 函数是一个非常好的黑盒例子，你知道你可以调用 prompt，再带上一个字符串，然后一个窗口就会出现，上面有这个字符串，一个文本输入框，以及一些按钮。但当你调用 prompt 时，你不需要知道执行了哪些指令才让那个窗口显示出来。它就是可以工作，知道这些足矣。如果那些在谷歌浏览器上工作的人，发现并修复了一个缺陷，影响了 prompt 的工作原理，你什么都不需要做就可以得到这个修复。如果那些人决定改进窗口的外观，你什么都不需要做就可以得到新的外观。

> **软件魔术大揭秘**
>
> 我多次说过使用他人的工作成果有多好，只要它能工作，你就什么都不需要知道。希望有人持不同观点：如果你什么都不需要知道，那你为什么读这本书呢？真相是，你确实需要了解你的框架，以避免你误用它。
>
> 要开车，你不需要知道发动机、差速器、内燃机，或者他们在雨刷液里放了什么让它那么高效。你甚至不需要知道如何换挡——自动挡汽车会帮你做。你只需要知道油门是干什么的，刹车是干什么的，以及什么时候该用哪一个。你需要知道 P、R、N、D 和 L 档，以及什么时候用什么档。如果你不花时间去了解汽车的大概结构，你就可能误操作。例如，你可能想要让车动起来，同时你知道 R 挡能让车动起来，结果你就一直倒着开车。你虽然达成了目标，但还有更好的方式。如果在不了解的框架上构建软件，你就很可能会犯类似的错误。

编程语言

编程语言是“抽象层次”的一个完美案例。计算机不理解编程语言；编程语言是为人类读写使用方便而设计的。没错，那些代码对你来说毫无意义，但至少计算机能理解它——好吧，计算机也不理解。对人类来说，用编程语言给计算机发送指令要简单许多（和写二进制码比）。这些指令仍然需要被转换成计算机能理解的东西（一堆 0 和 1），编程语言自己有一些指令来处理这一过程。

有很多编程语言供你选择，各自有其优缺点。有些语言有专门的用处（比如，R 语言几乎专门用在统计应用中），而另外一些语言则是通用的（比如，Java 和 C++几

乎可以用在任何地方）。用哪门编程语言取决于你要构建什么类型的软件应用。有些场景，你有很多语言可选（如果你要搭建一个网站服务器，你可以从 Java、 Python、Ruby、PHP、Perl、 JavaScript、 C#，以及其他很多语言中选择）。在另外一些场景，你几乎没有选择（如果你想做一个安卓应用，你只能用 Java，而如果你想做网页应用，你只能用 JavaScript）。下面比较了一些最流行的语言，这个清单可以作为参考，帮助你决定要花时间学哪门语言。

Bash

Bash 最常用于编写自动化命令行任务，比如文件系统操作。Bash 是系统管理员的主要工具之一。

- Linux 命令行（参见第 3 章“认识你的计算机”）中的每个命令都可以看作一个小的 Bash 程序，所以学习 Bash，上手相对容易，进阶相对难一些。
- 了解 Bash 可能不会让你找到一份工作，但要想成为系统管理员，你需要懂 Bash。而且，对于其他专业程序员来说，它也相当有用。通过自动化常用任务，Bash 可以让你非常高效（参见第 3 章）。

C

C 是一门通用语言，但它主要用于编写对速度要求高的应用，如果你想要写桌面应用，这是一个很好的语言。

- 目前很多流行语言的语法，都至少有一部分继承自 C 语言。C 的语法并不难懂，但我仍然觉得 C 是一个相对难学的语言，特别是作为第一门语言。
- 你仍然可以找到很多 C 语言的工作，但绝大多数情况下，同时学习 C 和 C++ 会让你得到更好的待遇。
- Windows 操作系统和很多 Linux 上的应用都是用 C 写的。

C++

C++和 C 很像，具备一些扩展的语言功能，包括 C++是面向对象的编程语言。

- 在高中和大学，C++是计算机科学入门课程的通用语言，C++课程会教大量的基础支持，但是对于初学者来说，它仍然是一门较难的语言。
- 有大量的 C++职位。
- 谷歌浏览器、火狐浏览器、微软 Word、Excel 及 PowerPoint 都是用 C++写的。

C#

C#，微软.NET 框架的一部分，是一门兼有 C++和 Java 特点的通用语言，可以用来搭建不同类型的应用，最主要用于 Windows 操作系统。如果你喜欢在 Windows 上工作，C#是很多类型任务的不二选择。

- 正如 C++和 Java 一样，C#对于初学者来说并不容易。不过，当你读完本书的时候，你就可以开始学习它了。
- C#几乎是.NET 框架里最流行的语言，而且有很多 C#的工作需求。
- Windows Phone 应用是用 C#编写的。

Java

Java 是一个流行的通用语言，很多学校教授 Java 作为计算机科学入门课程。它是业界最广泛应用的编程语言之一，几乎可以用于任何场景。

- Java 不是很容易学习，但是，考虑到它的流行性和实用性，还是有很多初学者学习——它还是比 C/C++容易一些。
- 如果你是一个优秀的 Java 开发者，你应该不愁找工作。
- 安卓应用和很多流行网站（包括 LinkedIn 的很大一部分）的服务端是用 Java 写的。

JavaScript

JavaScript 是一种 Web 编程语言，最初由网景公司（Netscape）发明并用于网景浏览器（Netscape Navigator）中。JavaScript 目前是一门也是唯一的一门用在每一个浏览器中的语言，每一个网站都要靠它提供交互体验。本书中，我们用它开发我们的 Chrome 扩展应用。如果你想要编写网站，你需要学习 JavaScript。（友情提醒，与普遍观点相反，JavaScript 与 Java 基本没什么关系，它们是完全不一样的语言。）

- JavaScript 语法很像 C 和 Java。JavaScript 算是比较容易上手，因为它是一门脚本语言，具有较少的数据结构和限制，然而，这也很容易导致写出非常差的 JavaScript 代码。
- Web 开发者处于高需求中，要成为 Web 开发者，你必须学习 JavaScript。
- Gmail、Twitter、Facebook、亚马逊，以及 LinkedIn 都在用 JavaScript。

Objective-C

Objective-C 主要用于写 Mac OS X 和 iOS 应用（参见下面 Swift 介绍），它是 C

语言的超集，意味着 C 能做的它都能做，而且还能做更多。

- Objective-C 有些难学。尽管它就是 C 和一些额外的功能，它的语法和约定都不同，当你接受了这些不同，那它就不比学 Java，C，或者 C++更难。
- 看到有多少 iPhone 手机了么？每个公司，要想做 iPhone 应用，就需要一个 Objective-C 开发者（至少目前是这样，参见下面 Swift 介绍）。
- iOS（iPhone/iPad）和 Mac OS X 应用（截止 2014 年 5 月）都是用 Objective-C 编写。

Perl

Perl 是一门通用语言，广泛用于大量的文本操作。就因为这个原因，Perl 是我学的第一门语言。在第 6 章“正则表达式”中，你会学习正则表达式，Perl 是这些正则表达式的祖师爷。如果你有大量的文本报告需要转换成其他更有用的东西，Perl 就是你需要的语言。

- Perl 学起来相对容易。它的语法类似 C，但它是一门解释性语言，执行代码相对容易，而且，Perl 是一门动态语言，你不需要完全了解数据类型就可以开始写代码，参见第 5 章“数据（类型）、数据（结构）、数据（库）”。
- 你仍然可以找到 Perl 的工作，但你也许该考虑学一下 Python 或者 Ruby。

PHP

PHP 是一门通用语言，从简单到复杂的 Web 服务器都可以编写。它的流行得益于其容易学习，并且方便连接数据库（见第 5 章）。如果你刚刚开始学习编程，并且想要学习搭建一个 Web 服务器，PHP 是一个好的选择。

- PHP 非常容易学习，但是奇怪的特性和缺陷让 PHP 开发有些让人抓狂。
- 有很多 PHP 的工作，但是 PHP 的流行程度正在降低，Python 和 Ruby 可能会是更好的选择。
- Facebook 的 Web 服务器是用 PHP 写的。

Python

Python 是为了更容易的读和写而被发明出来，它是一门强大的通用语言，读起来就像纯英语。Python 让你不用写很多代码就能做很多事情。它的在线社区也有相当的规模，所以如果你不知道怎么做的时候，可以找到很多支持。

- Python 是最容易学的语言之一。对于初学者，它是很棒的选择（很多计算机科学入门课程现在都用 Python 来教学）。
- Python 学的好，你会得到很多工作机会。
- 比较新的 Ubuntu 应用和 Instagram 的 Web 服务器是用 python 写的。

R

R 是一门专用语言，主要用于统计分析。如果你需要分析的数据集规模很大，以至于无法用电子表格来做，R 就是为此而生。因为 R 为统计分析做了优化，它能够快速和高效地处理那些工作。使用 R 来做其他事情可能不是个好主意。

- R 相对容易入门，特别是如果你有一些数学背景。如果你没有数学背景（就像我），在理解一些 R 的核心特性时，你可能会遇到一些困难。
- R 加上扎实的统计知识可以让你找到工作，但只会 R 可能没什么用。

Ruby

Ruby 是一门相对新的语言，被设计得非常简单和直观。Ruby 是一门通用动态语言，用于所有类型的应用，包括最广泛采用的 Web 服务器。Ruby 开发者都很喜欢 Ruby，他们是一个忠诚的群体，而且有非常强的社区支持。

- Ruby 设计得非常容易学习，非常合适作为第一门语言。
- Ruby 程序员的需求非常强劲，如果你学好 Ruby，会找到很多工作机会。

Swift

Swift，于 2014 年面世，是苹果发明的语言，用来替代 Objective-C，作为官方支持的 iOS 和 Mac OS X 开发语言。Swift 是一门通用动态语言，大家刚开始了解 Swift 及其特性，目前为止开发者看起来还是很喜欢它的。

- 作为一门动态语言，学习 Swift 要比 Objective-C 更容易一些。
- 随着时间推移，Swift 会成为 iOS 开发者的必备技能，如果你还不会 Objective-C，你应该开始学习 Swift。

VBA

VBA 这门语言用在微软 Office 产品中，用于任务自动化。尽管只在这个有限范围内，它依然是一门强大的语言。当我还是一名审计员的时候，我经常在 Excel 里使用 VBA。

- VBA 不是很难学，但相比当下很多流行语言，它的语法有点奇怪。
- 你可能不会找到很多 VBA 的工作机会，但如果你从事会计或者金融方面的工作，懂一些 VBA 会帮你节省很多时间，它甚至可能会帮你升职。

从源代码到 0 和 1

你不必弄明白你的代码是如何变成二进制码，但你得知道有这个过程，并且你应该理解其原因。一门编程语言本身就是一个软件：一组指令，处理用这门编程语言写的代码，将其转换成计算机能执行的东西。如果你写的代码不对，你的指令就可能在转换过程中丢失一部分，进而程序就不会按照你想要的方式运行。我曾经觉得，这太蠢了，简直荒谬，几百行代码无法工作，就因为我忘了一个右括号或者分号。直到我的代码必须在执行前被转换，让我理解了这事儿，如果我的指令写错了，编程语言就无法正确地将其转换成二进制码。

编译型语言与解释型语言：源代码何时变成二进制码

有些语言要求，在代码执行之前，你的所有代码都要执行编译；其他语言在代码执行时，解读每一条指令。事实上，编译型语言通常是由程序员在他的计算机上将程序编译好；当他需要分享（或者卖）这个软件时，他会分享编译好的 0 和 1，而不是源代码。

当你用编译型语言写程序时，你必须在修改源代码之后，再次运行程序之前，将程序编译好。用编译型语言工作有时候挺好，因为很多缺陷会在编译源代码时被发现。编译错误有时候让人沮丧，因为一个错误就会阻止编译你所有的代码，而不只是出错的那一行。

相反，解释型语言不需要编译，所以有关编译错误的爱恨情仇，你都体会不到。解释型语言需要个名为解释器的软件，用来处理源代码，每次执行其中一条指令。解释型语言写起来会快一点也容易一点，因为每次修改代码之后，你不必重新编译源代码了。编译可能会花一些时间，所以跳过这个步骤，可能会省下很多时间。

编译型与解释型语言的优缺点

编译型语言

优点

- 性能更好

- 有更好的集成开发环境
- 用户计算机不需要知道软件是哪种语言所写，因为程序已经是二进制码

缺点

- 需要较长的编译时间
- 针对不同处理器，必须采用不同的编译方式（PC 上的处理器和 Mac 上的工作方式不一样，所以代码需要在二者之上分别进行编译）
- 初学者开始时会有困难（要学更多工具，理解更多晦涩的错误）

解释型语言

优点

- 代码改动不需要重新编译
- 初学者容易学习

缺点

- 性能相对较慢
- 需要用户计算机安装解释器
- 有些缺陷难以捕捉，因为代码不需要编译

运行环境

你已经知道源代码需要转换成二进制码，计算机才能理解它，那么你可能已经猜到，每一种编程语言都有它自己的转换方式。你以前知道编程语言要安装吗？不安装编程语言（或者更准确觉得说，是编程语言的一种实现），源代码到二进制码的转换就不会发生，进而你的代码就不会执行。既然你在写代码，你就需要一个地方来执行代码——一个运行时环境。对于 JavaScript 来说，网络浏览器就是最常见的运行时环境。对于很多编程语言来说，集成开发环境也是运行时环境。对于一些语言来说，命令行就是运行时环境。

图 2.2 用一个很形象的比喻，来解释编译型语言、解释型语言，以及运行时环境。对于一架自动钢琴来说，源代码就是乐谱。乐谱就是一组指令，由一位作曲家写成，而其他作曲家都能理解。然而，自动钢琴不能理解乐谱。乐谱必须被“编译”成自动钢琴的打孔纸带，自动钢琴才能执行。如果作曲家改了最初的乐谱，这个纸带就必须重新编译。一旦编译完成，纸带可以插进自动钢琴，上面的音符就会被“执行”。如果乐谱没有编译，作曲家可以作为一个“解释器”，现场演奏这些音符，但作曲家

每次演奏这首歌时，都必须解释这些音符。最后，一个普通钢琴无法演奏这些纸带，因为它没有安装“运行时环境”。然而，只要有一个解释器（作曲家），一个普通钢琴就可以演奏这些乐谱。

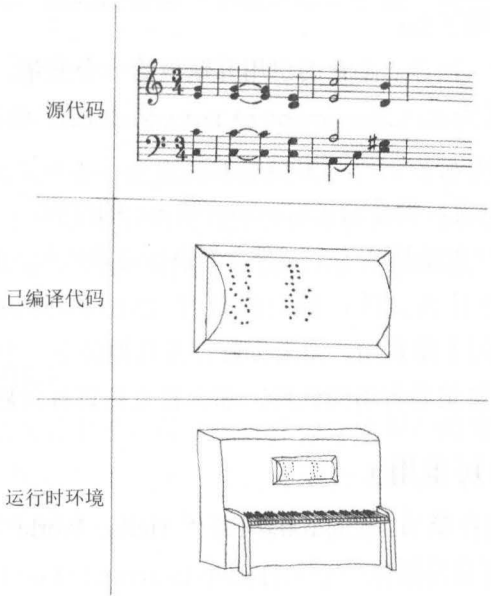

图 2.2 自动钢琴与计算机类比

处理器执行

软件生命周期的最后一步，就是执行。这时，你的指令已经被转换成 0 和 1，被送入计算机的处理器（第 3 章会有更多处理器的介绍）。处理器知道该怎么处理那些 0 和 1，它会执行你的指令。至此，你的程序跑起来了！

输入和输出

到目前为止，你已经看到一组人类可读的指令，先以源代码的形式存在，再被转换成 0 和 1，最后被计算机执行的全过程。很好，但是如果我们想让程序根据情况做些别的事呢？如果指令每次都做一样的事，可能用处不会很大。例如，想象一下，微软 Word 软件打开文件的源代码，就像清单 2.2 那样。

清单 2.2　打开 Word 文档的伪代码

```
var wordWindow = openWordWindow();
var fileContents = loadFile('C:\Documents\Intro.docx');
displayFile(wordWindow, fileContents);
```

这段伪代码相当直截了当：

1．打开 Word 窗口，保存一个它的引用并赋值给一个变量，名为 wordWindow。

2．加载文件，路径为 `C:\Documents\Intro.docx`，将其内容保存为一个变量，名为 `fileContents`。

3．在 wordWindow 窗口中显示 `fileContents` 的内容。

如果每次 Word 用户想要打开一个文件，都执行这段代码，你能看出其中的问题吗？不论用户想要打开什么文件，都只能打开 `Intro.docx`。如果你想要打开 `Intro.docx`，这段代码非常有用，如果你要打开其他任意一个 Word 文档，这段代码都完全没用。我猜你是想要有用的软件，那么我来告诉你问题在哪：输入。

输入让软件更实用（可重用）

你可能会想起我们在第 1 章遇到的问题“‘Hello, World!’ 写下你的第一个程序”。一开始，你的网页只是跟你一个人打招呼（`alert('Hello, Steven!');`，还记得吗？）。通过加入 prompt 那段代码，你在程序里引入了输入。不可否认，kittenbook 目前依然没什么用，但它现在至少能跟任何人打招呼。

我们回头看看微软 Word 那个例子。如何修改代码，让它可以打开任意文件，而不只是 `Intro.docx`？我们需要输入！在 `prompt` 的例子中，我们知道输入从哪来：提示对话框，但你可能通过好几种方式打开一个 Word 文档，而你又不想针对每一种场景重写代码。我们来创建一个函数解决这个问题。我们会在第 8 章“函数和方法”详细讨论函数，但现在，我们可以认为函数就是个有名字的迷你程序。函数可以接受输入，不在乎输入是从哪来的。这正是我们 Word 例子需要的（见清单 2.3）。

清单 2.3　使用函数来读取输入

```
function openWordFile(filePath) {
  var wordWindow = openWordWindow();
  var fileContents = loadFile(filePath);
  displayFile(wordWindow, fileContents);
}
```

我们的函数称作 `openWordFile`，我们还得给它一个文件路径（文件路径就像是文件在计算机上的住址）。这样，我们就可以通过双击一个文件图标打开 Word 文档，或者使用菜单里的文件→打开。编写可重用的代码，可以给你节省大量的时间，因为你只需要写一次，也只需要在一个地方修正问题。使用输入是编写可重用代码的重要部分。

输入从哪来

计算机程序的输入并不一定来自计算机用户，尽管他们是一个重要来源。输入可以来自很多地方，包括其他计算机程序、计算机的时钟、硬盘上的文件、互联网，以及外部设备。这些输入来源，有些可能让人摸不着头脑，用些例子来解释应该会比较清楚。

从其他程序得到输入

可能最常用的输入类型就是从其他计算机程序得到的输入。当你把一个文本文件保存到计算机上（比如 `kittenbook.js`），控制硬盘的软件（称为文件系统）接收到来自文本编辑器的输入。不同程序一直在你的计算机上相互交流。

从时钟得到输入

有些软件将时间作为输入。这类软件的一个例子，叫作 cron 任务（cron 来自 chronos，希腊语表示时间）。cron 任务在特定时间运行，比如一天的某个时间、一周的某个时间、一个月的某个时间、一年的某个时间等。它接收时间作为输入，用来确定什么时候运行。

从硬盘得到输入

每次打开你计算机上的文件时，你打开的程序正在使用来自硬盘的输入。下一节会讲到状态，对于来自硬盘的输入有多重要，你会有更深刻的理解。

来自互联网的输入

网络浏览器接收来自互联网的输入。网络浏览器收到 HTML（类似你在 `kittenbook.html` 里写的那种）作为输入，并将其显示在网页上。应用程序（比如 Dropbox 和印象笔记）使用来自互联网的输入，来保持你的文件和笔记同步。需要联网的移动应用，也使用来自互联网的输入。互联网是一个强大的（也是有趣的）

输入源。

来自外部设备的输入

当你移动鼠标，或者在键盘上打字，你就是在向计算机输入信息。你可以设置你的程序，去监听这一类输入。例如，你可以在鼠标进入屏幕的一块区域，或者点击某个特定按钮时，执行一段代码。你还可以接收如摄像头和麦克风之类的外部设备输入。Skype 用摄像头和麦克风作为输入，实现了视频通话。

软件如何获得输入

把输入传入你的代码这一过程可能有些挑战。这个过程足够常见，也足够困难，以至于有个专有名词描述它。执行这一过程被称为软件“接通”。从鼠标获取输入的难点，与从硬盘获取输入的难点完全不同，而两者与从互联网获取输入又完全不同。随着我们完善 kittenbook 这个 Chrome 浏览器扩展应用，我们会完成接通一些不同输入源的工作，到时你会看到这事有多棘手（值得）。

输出类型

输入很有用，但最终，我们想要我们的程序能够为我们做点事情。尽管有些例外，但大部分软件都应该有输出。输出可以有多种形式。有时输入是一个问题（“最近的快餐店在哪？”），而输出是一个答案；有时输入是杂乱的数据，而输出是条理清晰的数据；有时输入是一首歌的名字，而输出就是通过计算机扬声器播放这首歌。输出可能极其简单，比如在屏幕上显示一些文字；但它也可能非常复杂，比如播放一段视频，同时同步播放音频。一个计算机程序的输出，经常被用作另一个程序的输入，完全不是给人类使用的。

有时候，一个程序看起来不需要任何输出。有些程序存在的意义，是为了执行一些不以输出作为结果的任务。比如，当你发送一封电子邮件，你是用邮件发送程序执行了一个任务：获取电子邮件的正文，并将其通过互联网发送到对应的收件人。这个任务并不真的需要任何输出，但它确实有一个结果：电子邮件要么发出去了，要么没发出去。两种情况，你都想知道结果是什么。一个电子邮件程序，只执行发邮件的任务，而不把任务的结果输出出来，那就没什么实用价值。所以，即使你的程序只是执行一个任务，它仍然需要输出点什么，来传达任务的结果。

GIGO：垃圾进，垃圾出（Garbage In，Garbage Out）

输入和输出紧密相关；一个程序的输出，依赖于它接收到的输入。因此，如果一个程序收到垃圾数据作为输入，它就会产生垃圾数据作为输出。在 kittenbook 中，我们问用户的名字，然后（天真地）假设他们确实会输入他们的真名。如果不是他们的真名，我们假设他们至少会提供一个真正的名字，或者至少看起来是名字的文字。但是我们提供的文本框没有格式限制，我们的用户可以随意填写任何字符。他们可以填写他们的地址，他们最喜欢的电影对白，数学公式，甚至一些恶意代码。如果填了任何不是名字的内容，程序的输出就完全不通了，如：`Hello, Do, or do not. There is no "try"`。

种瓜得瓜，不可避免。如果你给了程序一个错误的输入，你就会得到错误的输出。然而，垃圾输入并不总是由于用户故意搞笑或者挖苦而产生，有时用户并不知道正确的输入是什么样子。有时一个程序的错误输出，成了另一个程序的垃圾输入。尽管你不能把低质量输入神奇地变成高质量输出，但你可以用一些技术，让你的程序更优雅地处理垃圾输入。

防御式编程

你知道先入为主的后果吗？通常都不太好。当你的程序接收到输入，你不该假设输入一定是对的。我们希望它是对的，但应该先检查一下。通常你的程序会期望输入是某种特定格式。就像你在第 5 章“数据（类型）、数据（结构）、数据（库）”和第 7 章“何时使用 `If, For, While`”中将要学到的，你可以写代码来检查输入确保是你想要的格式，如果不是，你可以修复或者显示一个错误。在不做太多假设的前提下编写代码，被称为防御式编程，这是避免垃圾输出的好办法，即使你接收到了垃圾输入。防御式代码是好质量软件的一个标志。

校验与清理

任何计算机程序接收用户输入的一个必要组成，就是输入校验。校验是防御式编程的一部分。例如，如果你构建一个程序，要求用户使用电子邮件地址，家庭住址和手机号注册，你要确保他们输入了真正的电子邮件地址，家庭住址和手机号。校验的第一步，是完整性检查：用户的电子邮件地址看起来是不是真的？是不是包含@符号？@符号两边是不是都有字符？电话号码是不是只包含数字，还是有其他字

母？这些检查是很重要的第一步，你会在第 6 章中学到具体怎么做。取决于这些信息对于你的程序功能有多重要，这个初步检查可能就足够了。

检查电子邮件地址看起来是不是合法，并不能告诉你这个地址是不是真正的邮件地址，甚至即使你发现这个地址是真正的邮件地址，你也不知道这个邮件地址是不是属于填写信息的用户。校验的下一步，就是确认。一旦你注册了一项新的服务，收到一封确认邮件，这就是软件在校验你的邮箱地址，确保它是属于你的。通过短信验证码来确认手机号也是一样的作用。

清理是防御式编程的一部分，负责清理输入，防止不好的事情发生。当一个恶意用户尝试提交输入，破坏你的程序，或者损害其他用户数据时，你可以（也应该）尝试在程序处理之前，清理这样的输入。当你写的程序只有你自己用时，校验和清理都不那么重要，但一旦你有其他用户，你需要开始考虑防御式编程。

状态

状态是软件工作原理的重要组成部分。软件中，状态的概念和其他语境中很接近。如果你要问我现在处于什么状态，我可能会用清单 2.4 中的 JSON 回答你。

清单 2.4　用 JSON 格式表示“我的当前状态”

```
{
  "isSitting": true,
  "consciousness": "drowsy",
  "mood": "restless",
  "heartRate": 73
}
```

看了我的状态，你可能会觉得我应该出去走走。类似的，软件也可以根据状态来决定它应该干什么。例如，Gmail 有一组键盘快捷键帮助用户更高效地管理邮件。一个最常用的快捷键（最少对我来说），是输入 g，然后输入 i，进入收件箱。如果我正在写邮件的过程中，我输入“给我几分钟”，Gmail 足够智能，不会把我带到收件箱去，尽管我在输入“give”时先输入了 g，然后输入 i。Gmail 有智能的使用状态：如果用户正在写一封邮件（状态），忽略键盘快捷键。

状态不会自动加入到你的程序中，程序也不是每个特征都重要。一切由你（程序员）决定你要追踪状态的哪些特征，也是由你决定保存哪些状态特征的副本。状态的一些特征是短期的（鼠标是否在按钮上？），几乎可以马上丢掉；另外一些状态

特征是长期的（用户是否登录），需要在用户使用软件的全部时间里都要记住；还有一些特征永远不能丢掉（用户的名字是什么？用户的邮箱地址是什么？）。不管一个特征需要保存多久，软件都需要维护一个状态副本，保持和真正状态同步。如果软件状态不能反映真实情况，就会有不好的事情发生。

给 kittenbook 添加状态

目前为止，我们的 kittenbook 软件没有状态，也不关心任何状态。我们先开始第一步，把状态放到我们的扩展应用中，给我们的“Hello”页面加入一些额外信息。你要把这些值添加到一个名为 `values.js` 的新 JavaScript 文件中，它应该位于一个名为 `js` 的新文件夹下。当你在这个文件夹下时，你可以把 `kittenbook.js` 移到这个新目录下，并将其重命名为 `prompt.js`。移动文件和重命名的目的是保持我们项目目录更有条理。一开始，把所有代码写在一个文件里合情合理，但很快就会变得难以管理（我知道，因为我试过）。一个条理清晰的目录结构，和条理清晰、命名合理的源代码文件，让开发更容易，也更有逻辑性。当下，你的 kittenbook 目录应该看起来像清单 2.5。

清单 2.5 kittenbook 目录的图形化展示

```
kittenbook/
├── js
│   ├── prompt.js
│   └── values.js
├── manifest.json
└── kittenbook.html
```

这些被添加到 `values.js` 文件的值有项目名称、版本号，以及当前时间（见清单 2.6）。前两个相当简单，但最后一个有些难办。

清单 2.6 用 `values.js` 给 kittenbook 添加状态

```
var projectName = 'kittenbook';
var versionNumber = '0.0.1';
var currentDate = new Date(); //创建日期对象，更多关于对象和日期对象的介绍，参见第 5 章。这个对象会被
                              //用来配置时间。
// currentTime 看起来和'2014-01-25 at 14:45:12'差不多
var currentTime = currentDate.getFullYear() + '-' +  // 设置年份
            (currentDate.getMonth() + 1)+ '-' +  // 设置月份
```

```
currentDate.getDate() + ' at ' +     // 设置日期
currentDate.getHours() + ':' +       // 设置小时（军事时间表示）
currentDate.getMinutes() + ':' +     // 设置分钟
currentDate.getSeconds();            // 设置秒
```

现在你可以更新 prompt.js，引入这些新值。我们把这些新值放到一个<p>标签，让它们在新的一行里显示，并且看起来小一点（见清单 2.7）。

清单 2.7　用 values.js 的值更新 prompt.js

```
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = '<h1>Hello, ' + userName + '!</h1>' +
                  '<p>' + projectName + ' ' + versionNumber +
                  ' accessed on: ' + currentTime + '</p>';
```

现在需要修改manifest.json来指向新的JavaScript文件。在"content_script"这节中修改"js"一行可以达到这一目的。注意你得在文件名前面添加 js/，因为新的文件都在 js 目录里。

清单 2.8　更新 manifest.json，引用新加的 JavaScript 文件

```
"js": ["js/values.js","js/prompt.js"]
```

现在你可以从扩展页面重新加载你的扩展应用。访问 https://www.facebook.com，填写你的名字，你应该能看到类似图 2.3 的内容。干得漂亮！你已经迈出了项目更有条理的第一步。这些改变可能现在看起来不太重要，但是将来（这里的“将来”，我的意思是“从第 5 章开始”）会让你的工作更高效。

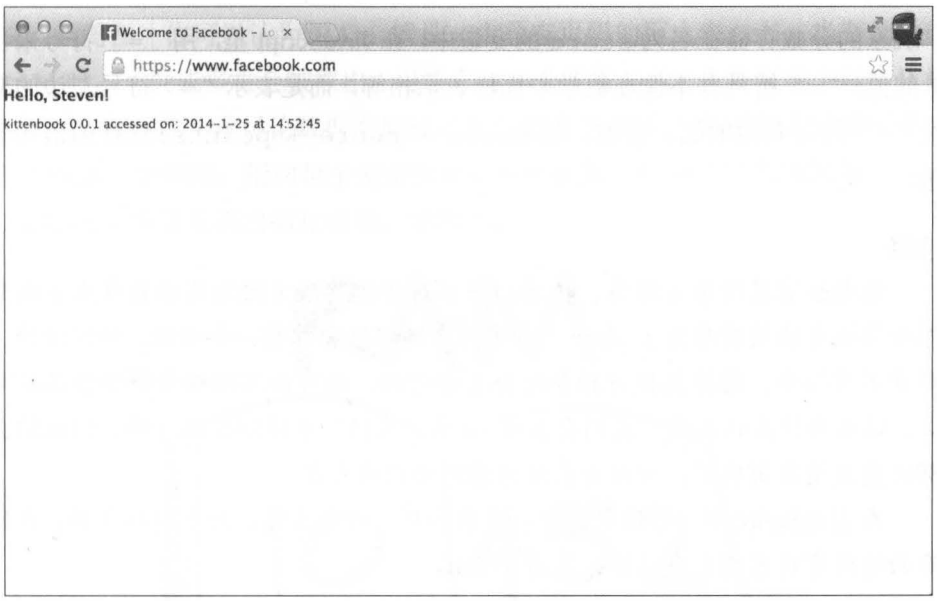

图 2.3 现在 kittenbook 显示了名字、版本，以及当前时间

内存和变量

下一章你就会学到，软件使用 RAM 存储它自己的信息。一个软件在运行中的所有信息都会经过内存：状态、变量，以及真正的指令，都存储在内存中。

变量

你已经知道，变量用来存储程序中随时会被使用的数据。变量是一个编程语言最重要的工具之一。变量真的是可变的，这就是为什么它这么有用。当你玩电子游戏时（比如“马里奥兄弟 3”），你的分数会被存在变量中。当你开始新的游戏，这个变量会被设置为 0，你每次得分，这个变量的值就会增加（见清单 2.9）。

清单 2.9 用变量存储分数

```
var score = 0;
// 通过传入的 points 的数量来增加分数
function increaseScore(points) {
  score = score + points;
}
```

玩家的分数存储在名为 score 的变量中。在 JavaScript 里，用“=”符号来设置变量的值。“=”符号并不像在数学中那样表示相等，而是表示“=”符号左边的变量接受“=”符号右边的值。否则，像 score = score + points;这样的语句就讲不通。

注释

编程语言是用英文所写，但他们并不是普通英语（因为英语存在太多歧义，而计算机无法处理歧义）。每个（有用的）编程语言都有一个功能，叫作注释。注释并不是指令，计算机执行指令时会忽略它们，你可以用注释来解释你正在做什么，以及为什么这么做。我们会在第 10 章“文档”中详细介绍注释，但我们会在那之前大量使用它们，所以你应该注意到它们的存在。

在 JavaScript 中，有两种注释：内嵌和块。内嵌注释以两个斜杠开始，在斜杠后面的所有内容都变成注释，见清单 2.10。

清单 2.10　JavaScript 的内嵌注释

```
// 内嵌注释
var score = 0; // 在真正代码之后的内嵌注释
// score = score + 10; 在真正的代码前面加上双斜杠，
//                     就将代码变成注释，
//                     这里 score 的值还是 0
```

块注释以一个斜杠和一个星号开始，以一个星号和一个斜杠结束，在这之间的所有信息都是注释（见清单 2.11）。

清单 2.11　JavaScript 的块注释

```
/* 注释从这里开始
var score = 10; score = score + 10;
注：这行代码不会被执行，
因为这是个注释
注释在此结束*/
```

变量存储

变量创建时，赋给它的值需要被存在某个地方。不同编程语言以不同方式处理变量存储，但他们都把值存在 RAM 中的某个地方。我们会在下一章更详细地讨论

RAM，但现在，你可以认为 RAM 就是一本具有页码的巨大空白书，当变量的值存在 RAM 时，一页或多页会被取出，用来存储那个变量的值。页数取决于程序认为那个值会有多大（第 5 章讨论数据类型时对此会有更多介绍）。与书后面的索引类似，程序会保存一个引用，指向每个变量值被存储的位置。更新这个变量的值，本质上就是更新这个变量关联的那几页书，见图 2.4。

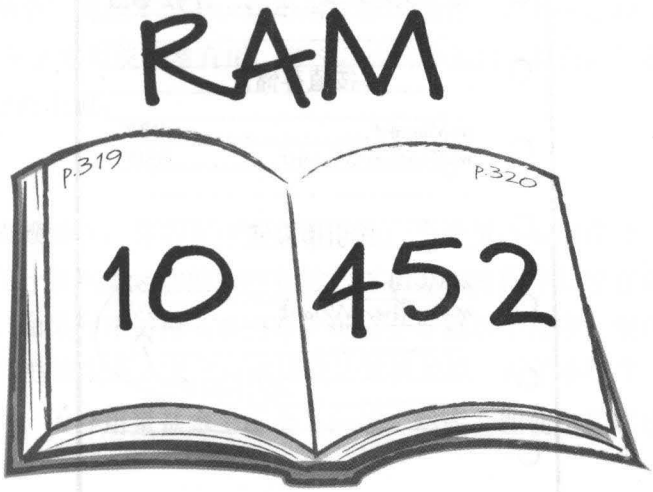

图 2.4 变量存储在 RAM 中的给定位置，就像书中的一页

视语言和场景而定，变量可能会以两种不同方式存储：按值或者按引用。清单 2.12 看起来更清楚一点。

清单 2.12 设置相等的两个变量

```
// 我体重 160 磅
var myWeight = 160;
// 我兄弟和我完全一样重
var myBrothersWeight = myWeight;
```

按值存储：变量复制机

如果变量按值存储，那么 myWeight 就会指向书的一页，其值为 160，并且 myBrothersWeight 会指向完全不同的一页，其值也是 160。第二个变量（myBrothersWeight）是第一个变量的复制（见图 2.5）。

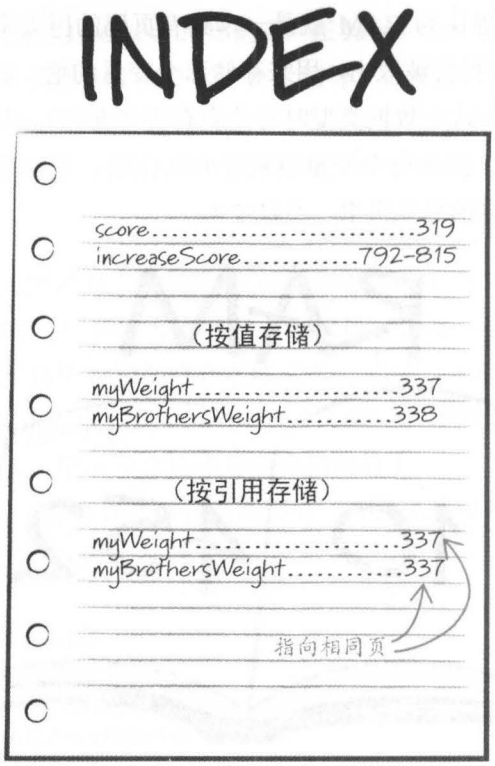

图 2.5　通过查看索引，你可以找到变量的值

按引用存储：玫瑰不叫玫瑰[1]

相反，如果一个变量按引用存储，那么 myBrothersWeight 就会指向 myWeight 的同一页。这个例子中，没有创建任何复制，而是两个变量都引用了书的同一位置。这个技术会节省空间，但它也会带来一个有趣的副作用（见清单 2.13）。

清单 2.13　设置两个相等的变量，然后修改其中一个

```
// 我体重 160 磅
var myWeight = {
  inPounds: 160
};
// 我兄弟和我一样重
```

1 出自莎士比亚，“What's in a name? That which we call a rose by any other name would smell as sweet.”，意思是：名称有什么关系呢？玫瑰不叫玫瑰，依然芳香如故。

```
var myBrothersWeight = myWeight;
// 我兄弟吃的有点多，长了 10 磅体重。
// 现在我多重?
alert(myWeight.inPounds);
```

如果两个变量指向书中的同一页，而你的代码更新了其中一个变量，这意味着两个变量的值都改变了。在上例中，`myWeight.inPounds` 会升高到 170，即使我兄弟是吃得多的那个。完全不公平。按引用存储可能有点让人困惑和讨厌（特别是这个例子），但实际上它可能非常有用，第 5 章你就会见到。然而，这种行为只有在变量是按引用存储时出现。

有限的资源

记住，软件依赖硬件；你的程序中存储的每一个变量，必须花费一些计算机硬件的物理空间。在计算机里，你的程序和其他正在运行的程序共享存储空间，而存储空间就那么多。如果程序占用太多内存，可能会导致这个程序，很可能是所有程序,运行缓慢。在有些情况下，甚至可能导致计算机关机。对于小程序，比如我们在本书中构建的那个，内存用量基本上不会是问题。然而，即使小程序也可能对内存有大影响。

内存泄漏

最开始写的程序之一，是一个 Perl 程序，用来重命名几百个 PDF 文件，需要在文件末尾添加一些字符。我用循环的技术（见第 7 章）来修改每个文件。在让它处理 300 个文件之前，复制两三个 PDF 文件来测试该程序。（作为一个通用法则，在把代码应用到真实场景前，在小规模数据上备份和测试代码，是个好主意。特别是当你和文件系统打交道时。）在这些复制上，一切都按预期方式工作，所以我就在几百个 PDF 的数据集上运行这个程序，程序运行的时间比预期的长，然后计算机就关机了。程序非常简短，而且看起来毫不起眼，以至于我不相信那是因为我的错误所致。我的意思是，我即使没怎么学过编程，却怎么可能错得如此离谱以至于整个计算机都关机？我重启了计算机，再次运行我的程序。然后它又一次关机。所以就是我的错。我不知道我做错了什么，所以我删除了这个文件，重新操作。

当程序用完一个变量，内存分配给存储变量的空间可以再次变为可用。即使程序不再需要这个变量，内存空间也永远不被释放时，程序员就制造了一个内存泄漏。

我当时就制造了一个内存泄漏。每次我的程序执行循环，都会使用更多的内存。看起来我还制造了一个无限循环（这个特别容易），所以那个简单的小程序就迅速占用了计算机的全部可用内存，以致计算机不得不关机。

总结

本章你学到了：

- 软件的定义
- 源代码，以及源代码如何变成二进制码
- 编程语言
- 输入和输出
- 软件可重用性
- 内存、变量，以及内存泄漏

写这章的时候，我很头疼，而且我每天的工作都要对付这些东西。我的头可能要炸了，但是你熬过来了，尽管你可能最后想要回来再读一遍本章。在本章你所学到的，是将要学习的其他所有编程知识的主要基础。

当你的头疼感觉好点时，在下一章，我们会学到：

- 计算机硬件如何工作
- 硬件和软件如何交互
- 文件系统是什么
- 如何使用命令行和文件系统交互

3

认识你的计算机

注

项目：用命令行来（1）创建一个新的 `release` 目录给 kittenbook，（2）把全部 JavaScript 连接成一个单独的文件，放在 release 目录中。

关于怎么使用计算机，你已经知道很多了。你可以开机，连接无线网络，还能编辑在线文档。你可能已经是 Outlook 或 Photoshop 的重度用户。不过，当你学习如何编程，你需要超越理解如何使用计算机上的软件，而去理解计算机如何工作。我们开始吧。

计算机很笨

计算机能下象棋，甚至每次打败象棋大师。不论你想去哪，计算机都能找到最便宜的机票；不论你想开车到哪，它都能确认躲避拥堵的最短路线。计算机很强大。所以现在可能是最佳时机，开启一个略显尴尬的主题：计算机很笨。对于它们运行的命令，计算机完全不理解。计算机也非常听话，我们告诉它们什么，它们就做什么，要满足两个条件：

1. 它们必须有执行指令的能力。例如，只是告诉计算机给你做一个三明治，并不意味着你就真的能得到一个三明治。大多数计算机不会做三明治。

2. 计算机必须理解你的指令。这就是编程语言发挥作用的地方。

计算机乐意执行它收到的全部任意指令，这是我们有差软件（质量低下又有害）存在的原因。计算机从来不会在收到一条指令时，想着“我不做这个！”计算机不会思考；它们只知道做。这也是我们有高质量和有用软件的原因。因为没有限制计算机做什么，就没有限制它能创造出什么。

计算机有魔力

你可能感觉有点矛盾。这里我要告诉你，计算机有魔力，尽管我刚刚告诉你计算机很笨。好吧，它们是笨；我不是在打退堂鼓。它们也有魔力，可能就是因为它们很笨。我们让计算机做什么，它们就做，而且做得很快。所以我们让计算机做任何我们能想到的事。而我们想出来了很多酷炫的东西：互联网、手机、视频会议、登月计划、找老朋友叙旧、结交新朋友、翻译……这个列表还在继续。

站在巨人的肩膀上

每一次创新和突破，都建立在上一次的基础上。程序代码曾经是一叠穿孔卡片上的一堆小洞。你必须会用二进制写指令（而且你必须保证你的穿孔卡片顺序正确）。一旦一张卡片放错了，整个程序都会挂掉。终于，程序员能用汇编写代码，这依然很难理解，但相比用 1 和 0，这要简单得多。汇编代码被翻译成 1 和 0。后来，其他编程语言被开发出来，它们写起来和理解起来要（相对）简单。你可以用英语写指令，其他代码会将你的代码神奇地转化成二进制，这样计算机就可以理解了。

学习编程最难的部分之一，就是明白你不能，也不会理解每一样事情。其他有些人已经知道如何把 C 代码转换成二进制码；你不需要理解二进制码，你也不需要理解 C 到二进制码的转换过程。再强调一次，这个概念叫作抽象。可能现在看起来没什么大不了的，但对于复杂性的抽象，使得利用他人的工作变得可能，这也让技术快速发展变得可能。

计算机内部

本节会讲一些计算机硬件的基础知识，与编写和执行代码的过程有关。这绝不是一个关于计算机硬件工作的完整描述。相反，本节只解释了你应该知道的，以便你能够开始编程。

处理器

CPU（中央处理器）是计算机实际执行指令的部分。这个很小、看起来不起眼的芯片，创造了几十亿美元的行业。在 F1 赛车的车身和职业自行车手的运动衫上都能看到它。它还是几次超级碗广告的主题。这是个大事。

CPU 是计算机的顶梁柱。其他我们讨论的硬件，都在某种程度上为 CPU 提供支持。

短期存储器

CPU 执行指令，随机访问存储器（RAM）是存储即将执行的指令的地方。把它看成短期记忆。如果我告诉你如何叠一个纸飞机，我的指令不可能存在你的手中，尽管你的手实际在执行我的指令。相反，指令被存在你的短期记忆中，通过你的手，一次执行一条。RAM 还存储数据，比如变量，这与程序的运行息息相关。我们会在第 5 章“数据（类型）、数据（结构）、数据（库）”深入探讨 RAM 的应用。

如果你像我一样，能够在大脑中同时保存 4 条指令，你可能想让这些指令重复几次。幸运的是，计算机的短期记忆比你的要好得多。计算机具有无比的专注力；它们会一直记住，直到你让它们忘掉，而且它们可以同时保存几百万条指令。然而，RAM 的能力有限，这就是为什么更多的 RAM 意味着你的计算机性能会更好；因为不用一直跑到后台去要更多的指令。

CPU 和 RAM 一起工作得非常紧密。它们在计算机里的位置很近，所以通信很快。它们没有活动部件，所以性能很快。它们是一个团队，但是，全部的指令是从哪来的呢？

长期存储器

计算机对于指令从哪里来并不挑剔。指令可以来自很多位置，包括命令行，直接下载进 RAM 的文件（比如网页上的 JavaScript 文件），或者硬盘中的文件。我们稍后讨论前两种方式，现在我们重点关注硬盘。

计算机硬盘是长期存储器。硬盘是存放你不想让计算机忘记的东西的地方——甚至计算机关机也不忘。把硬盘想象成一组文件，硬盘上有各种各样的文件。你可能有图片、演示稿，以及富文本文档。硬盘还存储包含指令的文件。当计算机执行这些指令时，会在硬盘上找到这些指令，读取它们，把指令发送给 RAM，CPU 会从这里读取指令并执行。

在编程时，你需要决定数据是存在短期存储器还是长期存储器里。存储在短期存储器中的数据，在计算机关机或者程序退出时会被清空，而长期存储器中的数据是永久保存的。短期存储器速度更快，但长期存储器通常有更多存储空间。把数据存储到哪里的决定通常比较容易做，但你需要知道，你确实是可以选择的。

使用计算机

不论你让计算机做什么，它都会去做——你只需要知道怎么告诉计算机。你现在拥有的计算机，能够做到曾经没人敢想的事情。但大部分计算机并没有发挥出全部潜能。它们只是做那些软件供应商告诉它们做的事。当我说“使用计算机”时，我不是要教你怎么使用鼠标、打开文件，以及打印一篇文档；你要开始像程序员一样使用电脑。

文件系统

计算机使用文件系统来存储和组织文件。你的硬盘、外部存储器、U 盘都是文件系统的一部分。当你创建一个文件，文件系统会在硬盘上给文件分配空间，然后创建一个指针，从文件名指向分配好的空间（见图 3.1）。根据文件类型，文件系统尝试猜测文件需要多少空间。如果文件数据量超过了最初分配的空间，文件系统会分配更多的空间。当文件被删除，文件系统并不移除这些数据；它只是溢出了指针，然后释放硬盘上的文件空间，让其他文件可以使用（见图 3.2）。这就是为什么删掉了的文件可以恢复回来：你只需要知道怎么找，以及在哪里找。

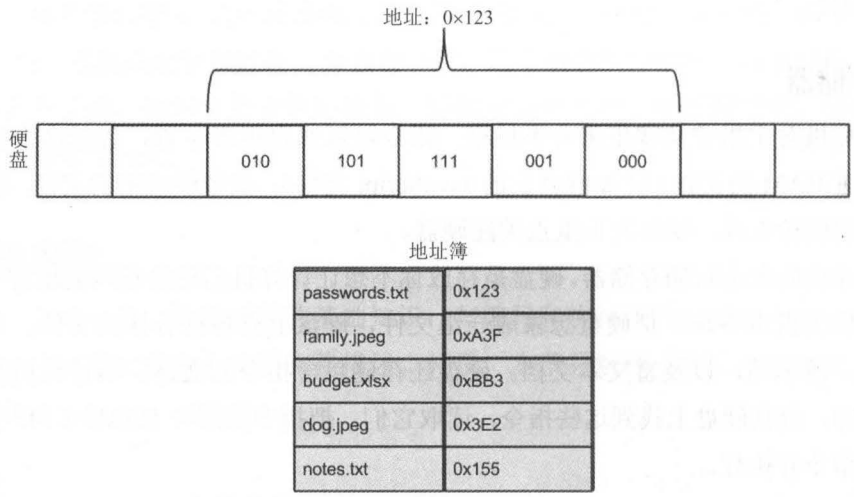

图 3.1　文件系统存储了一系列地址，将文件与硬盘中的位置关联起来

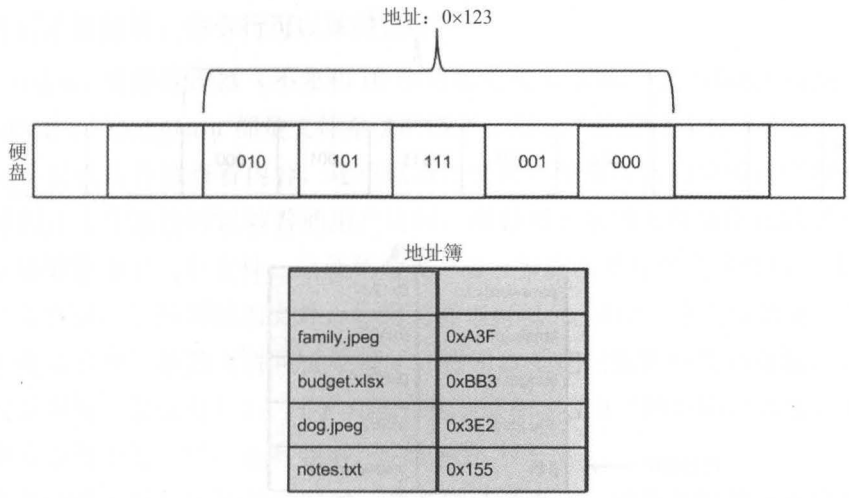

图 3.2 当 passwords.txt 文件被删除时，所有数据仍留在硬盘上，但是 passwords.txt 的记录从地址簿删除了

文件系统使用目录（或者文件夹）组织文件。一个好的文件夹结构，让你更容易找到你要的文件（例如，照片放在图片目录，文档放在文档目录），而且让程序更容易找到需要的文件。不过，有些用户不应该被允许修改（甚至读取）某些文件。出于这个原因，文件系统具备管理权限的能力，可以对一个文件或者一整个目录设置权限，权限可以设置成读、写和执行。

有时让两个不同的文件指向相同的数据会带来便利。你可以通过创建符号链接（类似 Windows 上的桌面快捷方式）来做到。创建出来之后，符号链接会像原始文件一样工作。如果你打开链接，会看到原始文件（见图 3.3）。如果你修改链接，原始文件也会改变。不过，如果你删除链接，你仅仅删除了指针——原始文件仍然存在。如果你删除原始文件，而没有删除链接，链接指向空白（见图 3.4）。

文件系统另一个隐藏的功能，是隐藏文件。在 Linux 和 Mac 上，这些被称为点文件（因为文件名以一个.开头）。点文件是看不到的，除非你显式地要求查看。通常点文件用来定义配置，并且由软件直接写入。当你在程序中选择用户首选项，那些首选项通常写入一个隐藏文件。用户也可以直接编辑那些配置。

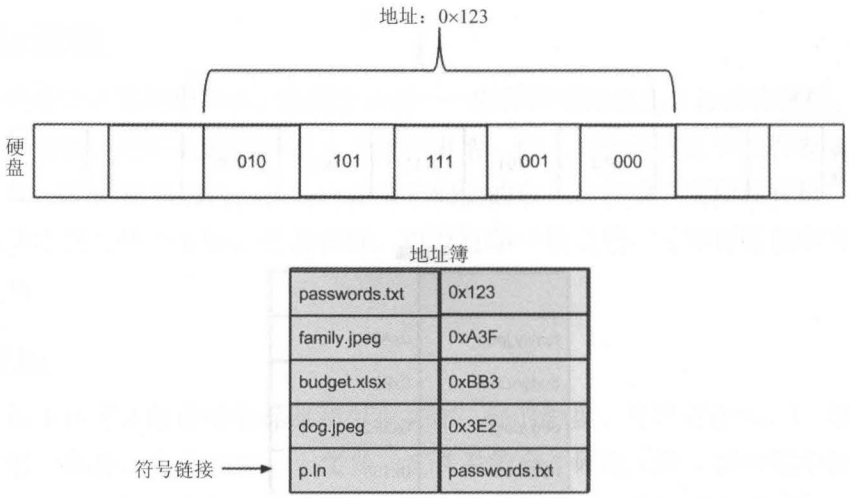

图 3.3　符号链接 `p.ln` 指向 `passwords.txt`，接着，`passwords.txt` 指向硬盘中的正确位置

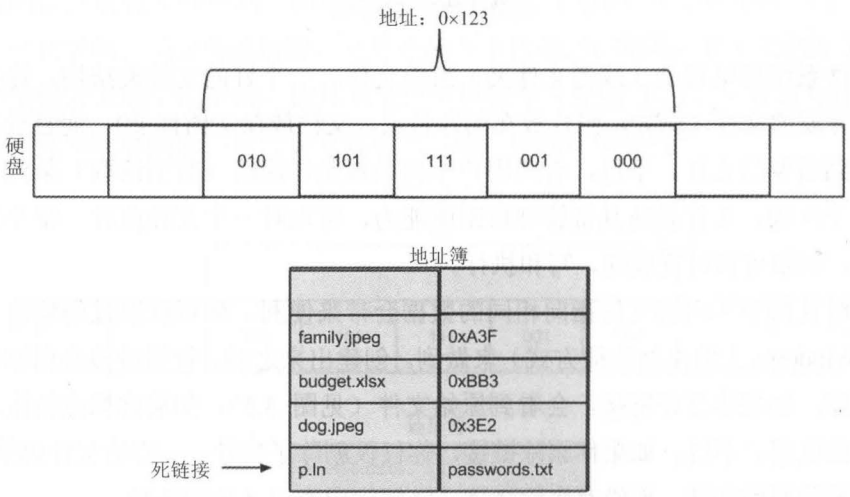

图 3.4　当原始文件删除，符号链接指向空白，这称为死链接

命令行：取得控制权

命令行是原始的，也是最强大的浏览和操控文件系统的方式。很多新手程序员抵触命令行，因为它看起来不够友善、复杂以及（嗯哼）书呆子气。你其实可以在不知道命令行的情况下走得很远，但最终你会需要它。早点学习使用命令行，你不会后悔。

界面能做的事，命令行可以做得更好

Windows 资源管理器（不要和 IE 浏览器混淆），Mac 上的 Finder 以及 Linux 上的 Nautilus 和 Konqueror 都是文件系统的图形化展示。它们把目录显示成小文件夹图标。你可以点击目录查看内容，还可以点击文件，使用默认的应用程序打开它们。这个界面让文件系统可以被普通用户访问，而且对于常用文件操作提供了易用性功能。如果要重命名一个文件，这就稍微难一点（答案不是打开文件然后点击另存为，输入新文件名，然后删掉旧文件，尽管这样也可以）。移动一个文件到另一个目录，可能不那么直接。删除文件可能很棘手（通常就是把它拖到回收站图标，但我找不到回收站图标，怎么办？）。对于大部分基础操作，GUI（图形用户界面）不错，但有些事它做得不怎么样，还有些事，它压根做不了。

当我还是个财务审计员的时候，最初的任务之一，就是复制前一年的所有审计文件到一个新的文件夹中，修改文件名，把结尾的 `2010` 改成 `2011`。

刚开始，我在图形界面做这个任务。复制文件到一个新目录相对简单。只要选中所有文件，按下 Ctrl+C 组合键，打开新目录，按下 Ctrl+V 组合键。不过有一个问题：我只想复制名字以 `2010` 结尾的表格文件。所以我用类型对文件排序，希望需要的文件会分成一组，不过当然，它们并没有。最终我只好一次复制 3、4 个文件，有时候一次 20 个文件，而不是我希望的一次复制 100 个文件。这还是简单的部分。所有文件都复制到新目录之后，我就得重命名它们。在 Windows 资源管理器中，单击一次会选中文件，然后单击一个选中的文件，就可以重命名这个文件。然而，双击文件会打开它。所以我要先点击一个文件，等一会，再点一次，然后按下右箭头键（让光标停在文件名的末尾），点一下退格键，再输入 1，然后把这个过程重复 100 遍。每 5 个文件中就得有一个，第二次点击前等的时间不够长，Windows 把我的点击识别成双击，然后我就得等着那台慢得要死的笔记本电脑（RAM 很小）把文件加载入 Excel；然后关掉这个文件，回到 Windows 资源管理器，重来一遍。10 分钟之后，我就想哭，想放弃，或者哭着放弃。之后我花了 30 分钟，重新学习如何写一个 Perl 脚本来帮我重命名文件。我执行了那个脚本，然后全部文件在 1 秒内就完成了重命名。

从那之后，我意识到，我只要用命令行就可以做全部事情，只要三个简单的命令（见清单 3.1），其中两个你会在下一节学到如何使用。

清单 3.1　使用命令行，1 秒重命名上百个文件

```
C:\Users\sfoote> cd audit\2010
C:\Users\sfoote\audit\2010> copy * ..\2011\*
C:\Users\sfoote\audit\2010> cd ..\2011
C:\Users\sfoote\audit\2011> rename *_2010* *_2011*
```

第一个命令复制全部文件到名为 2011 的新目录，第二个命令重命名文件，对于每一个以_2010 结尾的文件，都用 2011 替换 2010。按照计算机本来就应该的用法来使用计算机，我在 30 秒内就可以完成任务，而不是花一整个下午，做着让大脑迟钝的体力活。

基本命令

我对于命令行的最大恐惧，是觉得需要记住太多命令。尽管仍然对此感到恐惧，但我找到了一个比较小的基本命令集合，用这些命令，我完成了平时 80%的操作。在我们深入命令之前，先花点时间来看看命令行。

打开终端

第一步是打开一个终端。如果你已经知道怎么做，可以跳过本节。在 Linux 上，可以在“应用程序”→“系统工具”中找到终端。在 Mac 上，按下 Cmd+空格键，打开 Spotlight（见图 3.5）。在这里，你可以搜索“终端（Terminal）”。在 Windows 上，点击“开始”→“运行”，输入 cmd，然后按下回车键（见图 3.6）。

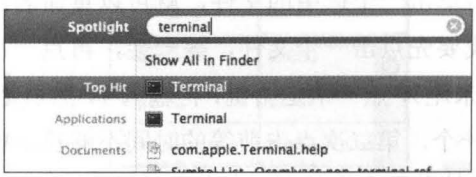

图 3.5　通过在 Spotlight 中输入 Terminal 打开 Mac 上的终端

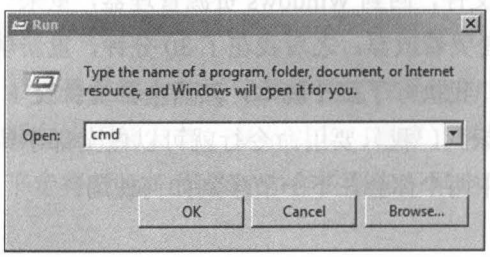

图 3.6　通过在运行弹出框中输入 cmd 打开 Windows 命令提示符

提示符

当你打开终端（或命令提示符），你不会看到太多东西——没有闪亮的按钮等着你去点。你只会看到看起来很神秘的一行字。恐惧，通常是因为不了解，所以让我们解密清单 3.2 中的这一行。

清单 3.2 提示符

```
sfoote@sfoote-mac:~ $
```

这一行有 4 个不同部分：用户名、计算机名、文件系统位置及模式。

1. 用户名：“sfoote”是我的用户名。就是我登录时用的名字。

2. 计算机名：“sfoote-mac”是我的计算机名。这是网络中其他计算机找到你的计算机的一种方式。

3. 文件系统位置：“~”是你的主目录，这是当你打开终端时的位置。这可能是这一行中最有用的一项。

4. 模式：“$”表明你是在正常模式下，具有正常的权限。不需要太在意这一项。

Windows 仅仅给你显示文件系统位置（见清单 3.3），再说一次，这是最重要的。Windows 还告诉你完整路径，而 UNIX 只给你当前路径（尽管你可以修改）。

清单 3.3 Windows 提示符

```
C:\Users\sfoote>
```

现在开始学习命令。学习命令的最佳方式就是使用它们。所以如果你还没在计算机前准备就绪，现在是时候了。在我们讲解这些例子时，在你自己的计算机上尝试把每一个命令都运行一下。你不需要输入#或其他跟在它后面的内容；那些只是注释，帮助你理解每一个命令都在做什么事。

```
pwd
```

在命令提示符中，有你所在的当前目录名，但是没有当前目录的完整路径。例如，我的命令提示符显示“sfoote@sfoote-mac:js $”，我知道我所在的目录叫 js，但是我可能有很多个叫 js 的目录，pwd（当前工作路径，present working directory 的缩写）可以告诉我当前目录的完整路径。Windows 下，显示的已经是完整路径，所以你不需要 pwd 命令，见清单 3.4。

清单 3.4　使用 pwd

```
sfoote@sfoote-mac:js $ pwd
/Users/sfoote/projects/website/static/js
```

当你首次启动命令提示符，你应该会在主目录中。对于接下来的练习，你需要到 kittenbook 目录中，但怎么到那个目录中呢？

```
cd
```

你不会总是待在主目录中。cd 可以让你切换目录。

> **绝对路径和相对路径**
>
> 当使用命令行在文件系统中导航时，你有两种方式来指定文件或目录的位置。这两种方式被称为绝对（完整）路径和相对（部分）路径。绝对路径通常有相同的开头（根目录），而相对路径是相对于当前工作目录。清单 3.5～3.10 更清楚地表明了这一点。

清单 3.5　cd 使用相对路径

```
sfoote@sfoote-mac:~ $ cd projects # 移动到项目目录
sfoote@sfoote-mac:projects $ pwd  # 现在运行 pwd，看看我们在哪
/Users/sfoote/projects
sfoote@sfoote-mac:projects $ cd kittenbook/js # 移动到 kittenbook/js 目录下
/Users/sfoote/projects/kittenbook/js
```

清单 3.6　cd 使用“..”移动到上级目录

```
sfoote@sfoote-mac:js $ cd ..        # 向上移动一级目录
sfoote@sfoote-mac:kittenbook $ pwd
/Users/sfoote/projects/kittenbook
```

清单 3.7　cd 使用“~”回到主目录

```
sfoote@sfoote-mac:kittenbook $ cd ~     # 移动到主目录
                               # “~”符号总是可以将你带回主目录
                               # 就像敲脚后跟（注：绿野仙踪中，敲三下脚后跟，就可以去任何地方）
                               # 只适用于 Linux 和 Mac
sfoote@sfoote-mac:~ $
```

清单 3.8 cd 使用“-”回到前一个目录

```
sfoote@sfoote-mac:~ $ cd -    # 移动回我们刚刚过来的目录中
                        # 之前我们在 kittenbook 目录下
                        # 所以我们会回到那里
                        # 只适用于 Linux 和 Mac
sfoote@sfoote-mac:kittenbook $
```

清单 3.9 复杂相对路径下的 cd

```
sfoote@sfoote-mac:kittenbook $ cd js   # 移动到 js 目录
sfoote@sfoote-mac:js $ cd ../../puppybook # 向上移动两级目录
                          # (../../), 然后从哪里到
                          # puppybook 目录
                          # 你需要创建(mkdir)
                          # puppybook 目录才能工作
sfoote@sfoote-mac:puppybook $ pwd
/Users/sfoote/projects/puppybook
```

清单 3.10 完整路径下的 cd

```
sfoote@sfoote-mac:puppybook $ cd /Users/sfoote/projects/kittenbook/js
sfoote@sfoote-mac:js $ pwd
/Users/sfoote/projects/kittenbook/js
sfoote@sfoote-mac:js $ cd ~/projects/kittenbook # 将主目录作为
                                                # 起始点
sfoote@sfoote-mac:kittenbook $ pwd
/Users/sfoote/projects/kittenbook
```

注意

注意当你在命令行中写路径时，试试按 Tab 键（在 Linux 和 Mac 中，你可能需要按两次）来自动补全正在输入的内容，或者查看一个可选项列表。

ls (dir)

用图形界面的文件浏览器时，我能看到当前你目录下的全部文件和目录。你可以用 ls（或者 Windows 上的 dir）在命令提示符中显示同样的列表。这个命令经常和 cd 一起使用，以帮助文件导航。见清单 3.11。

清单 3.11 使用 ls

```
sfoote@sfoote-mac:kittenbook $ ls
```

```
js   manifest.json  kittenbook.html
sfoote@sfoote-mac:kittenbook $ ls -a   # 列出全部文件和目录
                               # （包括隐藏文件）
                               # 你可能不会看到.git
                               # 我会在第15章解释什么是Git
. .. .git js manfiest.json kittenbook.html

sfoote@sfoote-mac:kittenbook $ ls js   # 列出js目录下的全部
                               # 文件和目录（你可以使用任意路径，
                               # 相对的或者绝对的，
                               # 这个命令都可以工作）
prompt.js values.js
```

你有没有注意到第二个命令中的“-a”？这是你首次遇到命令的参数标记。标记能改变命令的工作方式。有时候你会在标记之后加上一些文字，来进一步定制命令的工作方式。后面你会看到更多的标记。

cp（copy）

复制文件和目录，在命令行中真的非常容易（见清单3.12）。

清单3.12　使用cp

```
sfoote@sfoote-mac:kittenbook $ cd js
sfoote@sfoote-mac:js $ cp prompt.js new_prompt.js
```

cp命令接受两个参数，第一个是指向你要复制的文件的（完整或相对）路径，第二个是新文件的路径。第二个参数中的路径可以包含一个文件名，如果你想要重命名被复制的文件。小心，如果新文件已经存在，运行cp会覆盖那个文件，没有提示。清单3.12展示了prompt.js文件被复制到一个新文件，名为new_prompt.js。

清单3.13复制了整个js目录到js_copy。这说明我能够复制整个目录，并且能修改新目录的名字。-r标记（即递归，*recursive*）在复制目录中是必需的。

清单3.13　使用cp复制目录

```
sfoote@sfoote-mac:js $ cd ..
sfoote@sfoote-mac:kittenbook $ cp -r js/ js_copy
sfoote@sfoote-mac:kittenbook $ ls
js  js_copy  manifest.json  kittenbook.html
```

mv (move)

mv 命令和 cp 很像，不同之处在于它会删除原始文件（见清单 3.14）。

清单 3.14 使用 mv 移动文件

```
sfoote@sfoote-mac:kittenbook $ mv kittenbook.html js/      # 将 kittenbook.html
                                                  # 移动到 js 目录下

sfoote@sfoote-mac:kittenbook $ ls
js  js_copy  manifest.json
sfoote@sfoote-mac:kittenbook $ mv js/kittenbook.html . # 将 kittenbook.html 移回来
```

你可以使用 mv 移动文件，你还可以用它来重命名文件（见清单 3.15）。

清单 3.15 使用 mv 重命名文件

```
sfoote@sfoote-mac:kittenbook $ cd js
sfoote@sfoote-mac:js $ mv new_prompt.js prompt_copy.js
sfoote@sfoote-mac:js $ ls
prompt.js  prompt_copy.js  kittenbook.html  values.js
```

rm (del)

计算机程序员不喜欢多按不必要的按键，所以删除文件的命令是 rm，即 *remove* 的缩写。Windows 下的命令（del）要好猜一点。这个命令会删除一个文件或者目录（分别见清单 3.16 和清单 3.17）。不过，用这个命令要小心，因为没有撤销按钮。

清单 3.16 用 rm 删除文件

```
sfoote@sfoote-mac:js $ rm prompt_copy.js    # 删除 prompt_copy.js
sfoote@sfoote-mac:js $ ls
prompt.js    kittenbook.html  values.js
```

清单 3.17 用 rm 删除目录

```
sfoote@sfoote-mac:js $ cd ..
sfoote@sfoote-mac:kittenbook $ rm js_copy # 删除 js_copy 目录。
                        #这样不能工作，因为 js_copy 是一个目录。
rm: js_copy/: is a directory
sfoote@sfoote-mac:kittenbook $ rm -rf js_copy # 再试一次，添加两个标志位
    # 'r' 表示递归；删除这个目录以及其中的任意目录
    # 'f' 表示强制；不需要确认，直接删除
```

mkdir (md)

如果你能删除目录，就必须也能够创建它们。mkdir 会根据给定的路径创建一个新目录。如果你到这里实际上没有跟上，你现在就要想想，因为我们要开始做一些对于 kittenbook 扩展的工作来说很重要的修改。清单 3.18 和 3.19 展示了如何用这个命令创建一个新的 release 目录。

清单 3.18 使用 mkdir 创建目录

```
sfoote@sfoote-mac:kittenbook $ mkdir release # 创建release目录
sfoote@sfoote-mac:kittenbook $ ls
js   manifest.json  release
```

清单 3.19 使用 md 创建目录

```
C:\Users\sfoote\projects\kittenbook> md release
C:\Users\sfoote\projects\kittenbook> dir
Directory of C:\Users\sfoote\projects\kittenbook
04/24/2014 01:27 PM     <DIR>              js
04/24/2014 01:27 PM                324     manifest.json
04/24/2014 01:27 PM     <DIR>              release
        1 File(s)                324 bytes
        2 Dir(s)             XXX bytes free
```

cat (type 或 copy)

在 Linux 和 Mac 上，cat 常常用来串联文件，但是你还可以用它来快速查看单个文件的内容。cat 会把它的输出发送到你指定的位置。如果不指定，输出就会打印到屏幕。几乎所有命令行的命令都会以这种方式处理输出。事实上，这被称为“标准输出”，或 STDOUT。最大的优点是，你可以拿到这个输出，并把它放到任何你想要放的地方。你可以用这个输出当作另一个命令的输入，或者你可以把输出写入到一个文件中。

首先，我们来看清单 3.20，用 cat 查看两个文件的输出。

清单 3.20 用 cat 查看两个文件的串联输出

```
sfoote@sfoote-mac:kittenbook $ cd js
sfoote@sfoote-mac:js $ cat values.js prompt.js
var projectName = 'kittenbook';
var versionNumber = '0.0.1';
```

```
var currentDate = new Date();      // 创建日期对象，更多关于对象和
                                   // 日期对象的介绍，参见第 5 章
                                   // 这个对象会被用来配置时间
                                 // currentTime 应该是类似“2014-01-25 at 14:45:12”的样子
var currentTime = currentDate.getFullYear() + '-' +  // 设置年
                  currentDate.getMonth() + '-' +     // 设置月
                  currentDate.getDate() + ' at ' + // 设置日期
                  currentDate.getHours() + ':' +     // 设置小时（军事时间）
                  currentDate.getMinutes() + ':' + // 设置分钟
                  currentDate.getSeconds();          // 设置秒
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = '<h1>Hello, ' + userName + '!</h1>' +
                        '<p>' + projectName + ' ' + versionNumber +
                        ' accessed on: ' + currentTime + '</p>';
```

现在让我们将输出重定向到一个新文件，名为 `main.js`。我们还要把这个文件放到刚刚创建出来的 `release` 目录中。重定向输出到一个文件的方法，是使用大于号（>），使其位于命令和表示输出存放位置的文件名之间。如果输出文件不存在，会被创建出来。不过要小心使用，因为如果这个文件存在，它的当前内容会被命令的输出全部覆盖——再次强调，没有撤销按钮。

清单 3.21 使用 cat 串联两个文件并输出到新文件

```
sfoote@sfoote-mac:js $ cat values.js prompt.js > ../release/main.js
sfoote@sfoote-mac:js $ cat ../release/main.js
var projectName = 'kittenbook';
var versionNumber = '0.0.1';
var currentDate = new Date(); // 创建日期对象，更多关于对象和
                              // 日期对象的介绍，参见第 5 章
                              // 这个对象会被用来配置时间
// currentTime 看起来和“2014-01-25 at 14:45:12”差不多

var currentTime = currentDate.getFullYear() + '-' +        // 设置年
                  (currentDate.getMonth() + 1) + '-' +   // 设置月
                  currentDate.getDate() + ' at ' +  // 设置日期
                  currentDate.getHours() + ':' +     // 设置小时（军事时间）
                  currentDate.getMinutes() + ':' +  // 设置分钟
                  currentDate.getSeconds();          // 设置秒
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = '<h1>Hello, ' + userName + '!</h1>' +
                        '<p>' + projectName + ' ' + versionNumber +
                        ' accessed on: ' + currentTime + '</p>';
```

在 Windows 中，`type` 或者 `copy` 可以做串联文件的工作（见清单 3.22）。

清单 3.22 用 type 和 copy 在 Windows 上串联文件

```
C:\Users\sfoote\projects\kittenbook> type values.js prompt.js > ..\release\main.js
C:\Users\sfoote\projects\kittenbook> copy /b values.js+prompt.js ..\release\main.js
```

grep (findstr)

如果你需要找到一个包含某些特定文本的文件，grep 可以帮你做这件事。grep 可以搜索特定文本，如 Steven，并且它还能搜索模式，如 20XX（对应 2000、2002，以此类推）。见清单 3.23。

清单 3.23 使用 grep 来查找 kittenbook 中的文本

```
sfoote@sfoote-mac:js $ cd ..
sfoote@sfoote-mac:kittenbook $ grep -r "kittenbook" * # 在项目目录下的
                                                      # 所有文件中
                                                      # 搜索文字“kittenbook”
js/kittenbook.html:    <script    type="text/javascript"    src="kittenbook.js"></script>
js/values.js:var projectName = 'kittenbook';
manifest.json: "name": "kittenbook",
```

再次强调，学习这些命令的最佳方式，就是去实践。在一开始，可能要同时打开命令提示符和图形文件浏览器，以便你能够看到一种工具如何影响另一种。过不了多久，你就会看到命令行有多快。

总结

本章你学到了：

- 当你学习编程时，你的巨大潜力
- 计算机工作的细节
- 计算机如何使用文件系统来组织数据
- 如何使用命令行和文件系统交互

下一章中，你将学到：

- 自动化的编程任务
- 哪个任务更适合自动化
- 如何自动化 kittenbook 的任务

4 构建工具

注

项目：使用 Grunt 构建你的 Chrome 扩展、连接文件，以及进行代码质量检查。

前面两章的内容很多，也很难，希望不会无聊。你已经学了很多关于计算机如何工作以及软件如何工作的知识。那些章节讲了很多理论，较少实践例子，但你还是撑过来了，所以，恭喜你！干得不错！

和几乎每一种工作一样，编程也有很多重复性工作。代码需要编译，文件需要从一个目录移到另一个目录，测试需要运行，测试数据需要创建和更新，支持代码需要编译和执行等。程序员每次执行这些任务中的一项时，必须遵照同样的指令。程序员有一些输入（例如，某个文件要复制到另一个目录），执行一些指令，产生一些输出（文件成功复制到另一个目录）。这些看起来都很像第 2 章“软件如何工作”讲的，程序员在做的是软件工作。

程序员相比其他职业有一个优点，就是程序员要构建软件，而软件正是被用来消除（或者至少是减少）这些重复性工作的。构建会帮助自动化这些工作。具体来说，构建工具会从各种软件源代码文件中构建出“准备就绪”的软件。在复杂的软件项目中，构建软件意味着以正确的顺序编译正确的软件。房屋工程承包商会构建完美的穹顶、完美的墙、完美的地基，以及完美的储物柜，但如果他尝试在墙和地基就位之前就搭建屋顶，则不会得到完美的屋子。即使你的源代码没有错误，如果构建顺序错了，软件也不会工作。

（几乎）全部自动化

你不需要记住全部源代码的编译顺序，因为你可以使用构建工具来处理这些重

复性任务。不过为什么要提编译呢？JavaScript 甚至不需要编译步骤，但我们还是要在我们的项目中使用构建工具，因为大量的重复性任务可以被自动化。你能自动化的越多，你就有越多时间真正花在写代码上面。为构建工具写代码可能要花一些时间，但修建横跨美国的铁路也要时间；这两个都是很好的投资，在一开始花少量的时间和精力，从长远来看会给你节省大量的时间。项目每进行一天，你就会获得更多收益。

安装 Node

我们要使用一个叫作 Grunt 的软件作为构建工具。Grunt 是一个 JavaScript 框架，能够帮你做 grunt 任务（懂了么？）。你可能还能记起在第 2 章中，JavaScript 执行环境通常是 Web 浏览器中的一个网站。Grunt 不是网站，不在 Web 浏览器中执行，而是在命令行执行，所以尽管你有办法在计算机上运行 JavaScript 代码，但你可能没办法在命令行运行 JavaScript。

你需要安装一些软件，能够在命令行环境中执行 JavaScript。我们要用的这个软件叫作 `Node.js`。安装 `Node.js` 实际上非常简单：

1. 打开 http://nodejs.org/。
2. 单击写着“INSTALL”的巨大按钮，见图 4.1。
3. 点击下载的文件，见图 4.2。
4. 遵照安装文件提示的步骤（见图 4.3），使用默认配置。

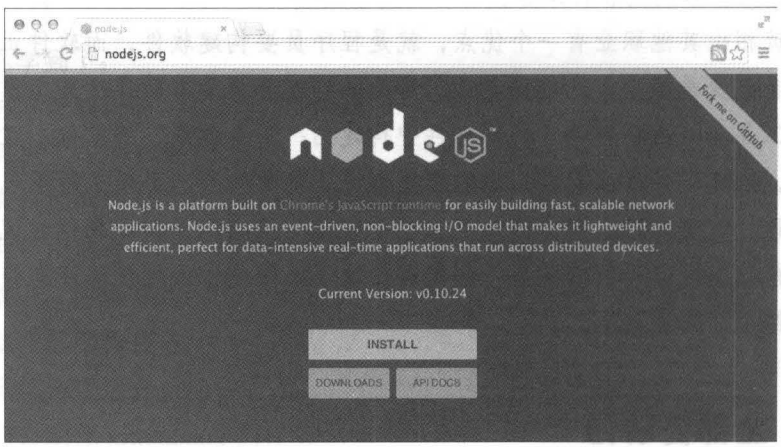

图 4.1　单击安装（INSTALL）按钮

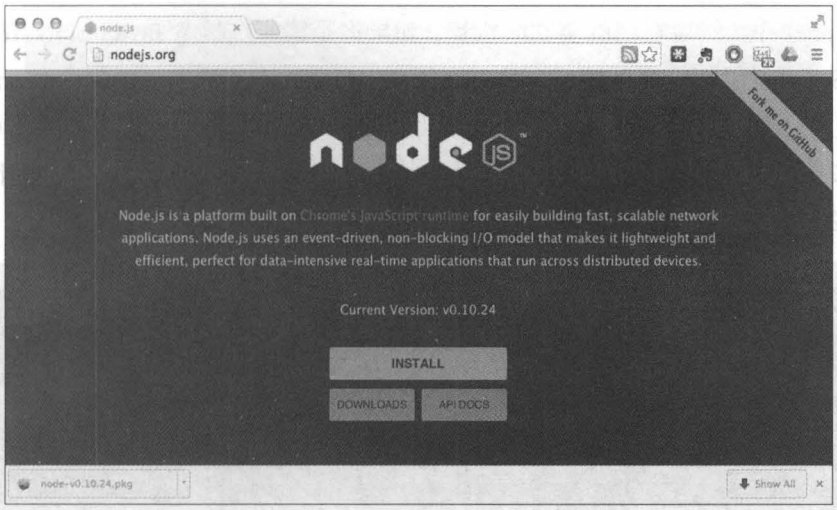

图 4.2 单击下载的文件，开始安装

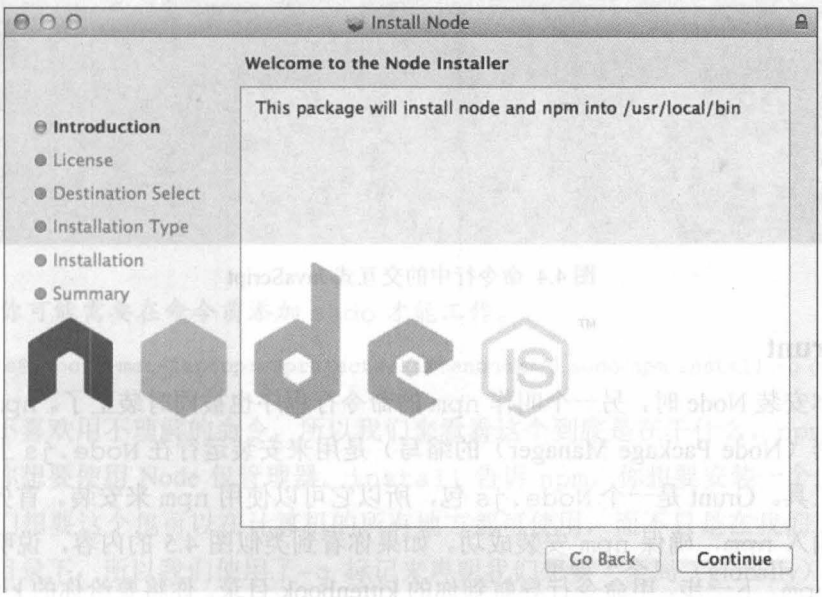

图 4.3 按照提示安装 Node.js

要检查安装是否成功，你需要打开一个命令行窗口。如果你不记得怎么做，回头看看第 3 章。当你有一个开着的命令行窗口，输入 **node**，然后回车。你应该可以看到像图 4.4 那样的信息，这是 Node.js 的交互模式。你可以一次一行地输入

JavaScript 命令，然后它们就会立即执行。如果你是第一次学习 JavaScript，在试验一些 JavaScript 特性时，交互模式会非常有用。很多其他动态语言（包括 Python、Perl、Ruby 和 R）都有类似的交互模式。你可以使用 Node.js 的交互模式（以及 Chrome 浏览器中的类似功能）来学习编程语言的特性。目前，需要知道的重要事情是，如果你可以使用交互模式，说明你成功地在计算机上安装了 Node.js。

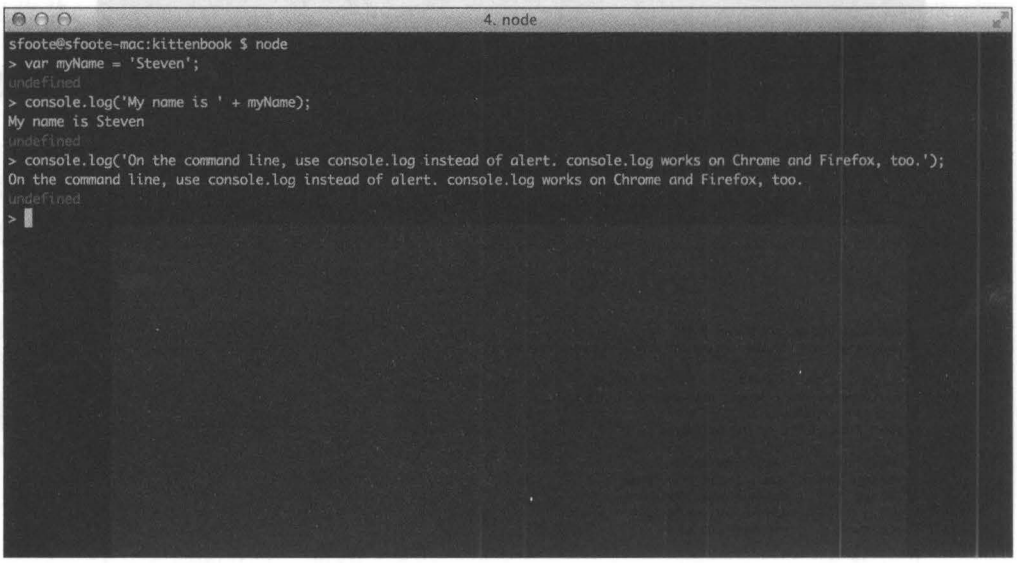

图 4.4　命令行中的交互式 JavaScript

安装 Grunt

当你安装 Node 时，另一个叫作 npm 的命令行程序也被同时装上了。npm（Node 包管理器（Node Package Manager）的缩写）是用来安装运行在 Node.js 上面的软件包的工具。Grunt 是一个 Node.js 包，所以它可以使用 npm 来安装。首先，在命令行里输入 **npm**，确保 npm 安装成功。如果你看到类似图 4.5 的内容，说明你已经装好了 npm。下一步，用命令行导航到你的 kittenbook 目录。你将要给你的 kittenbook 项目安装 Grunt。不过，首先你需要安装 Grunt 命令行工具，见清单 4.1。

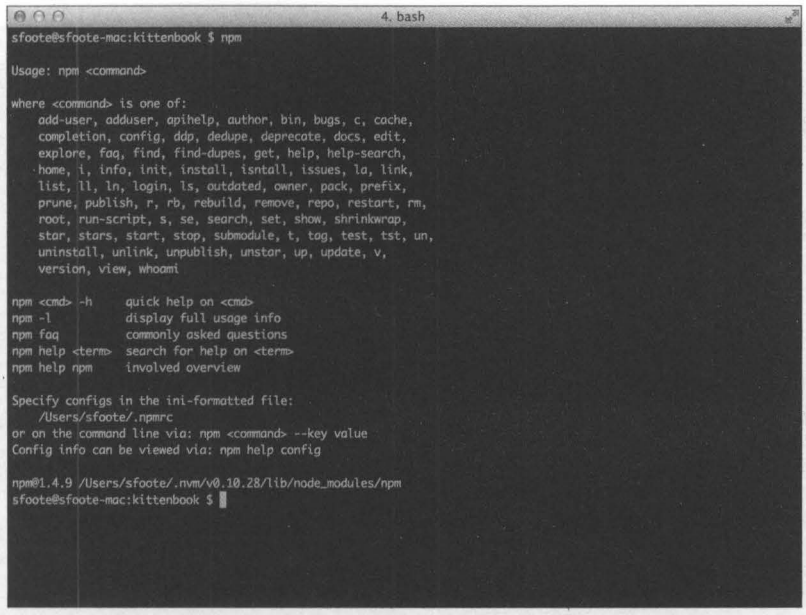

图 4.5 如果 npm 安装正常，当你在命令行运行 npm 时应该可以看到类似的信息

清单 4.1 安装 Grunt 命令行工具

```
sfoote@sfoote-mac-laptop:~/projects/kittenbook/ $ npm install -g grunt-cli
```

> **注**
>
> 你可能需要在命令前添加 sudo 才能工作。
>
> ```
> sfoote@sfoote-mac-laptop:~/projects/kittenbook/ $ sudo npm install -g grunt-cli
> ```

我不喜欢用不理解的命令，所以我们来看看这个到底是在干什么。npm 告诉计算机，你想要使用 Node 包管理器。install 告诉 npm，你想要安装一个包。在这里，我们想要这个包可以在计算机的所有地方都可使用，而不只是在我们运行命令的这个目录下，所以我们使用了-g 标记来声明我们想要“全局（globally）”安装。命令的最后一部分是这个包的名字，grunt-cli。

当 grunt-cli 安装完成，你需要定义你想要 Grunt 执行什么类型的任务。对于我们这个项目，我们要复制和链接 JavaScript 文件，以及用 JSHint 检查 JavaScript 代码质量。

回顾一下你创建的 manifest.json 文件，让你的代码以 Chrome 扩展的方式

工作。我们需要一个类似的文件，让 Grunt 为我们的项目工作。这个文件叫作 package.json，它用来告诉 npm 关于我们项目的信息，以及我们项目需要哪些包。我们项目需要的包称为依赖。package.json 文件（你需要在 kittenbook 目录下创建这个文件）需要我们项目的名字、版本号（和你在 manifest.json 里定义的一样），并列出依赖它们的版本号，见清单 4.2。

> **依赖**
>
> 如果你的软件只有在其他软件可用的情况下才能工作，那么其他软件就被称为依赖。依赖很棒，因为它们让你利用其他人的工作，而不用自己做。当你的依赖也有依赖时，事情变得有些棘手；依赖的依赖也可能有依赖，子子孙孙无穷匮也。像 npm 这样的包管理器会为你找到这个依赖链中的每一个依赖。查找全部依赖的过程称为依赖解析。

清单 4.2　kittenbook 的 package.json

```
{
  "name": "kittenbook",
  "version": "0.0.1",
  "devDependencies": {
    "grunt": "~0.4.2",
    "grunt-contrib-concat": "~0.3.0",
    "grunt-contrib-jshint": "~0.6.3",
    "grunt-contrib-copy": "~0.5.0"
  }
}
```

当你在 package.json 文件中填好了名字（kittenbook）和依赖后，可以在项目的根目录（kittenbook）下执行 npm install，然后 npm 就会下载并安装你的全部依赖，见图 4.6。

之后我们会配置 Grunt 来处理我们的构建任务。现在，确定 Node 和 Grunt 都成功安装在我们的 kittenbook 项目中就够了。如果你在安装 Node，Grunt 或任何 Grunt 依赖的过程中遇到什么问题，查看 http://gruntjs.com/getting-started 获取更多详细的安装指导。如果还是有问题，在互联网上搜索一下（更多关于如何搜索编程相关的答案，参见第 13 章“授人以渔：如何用一生学习编程”。

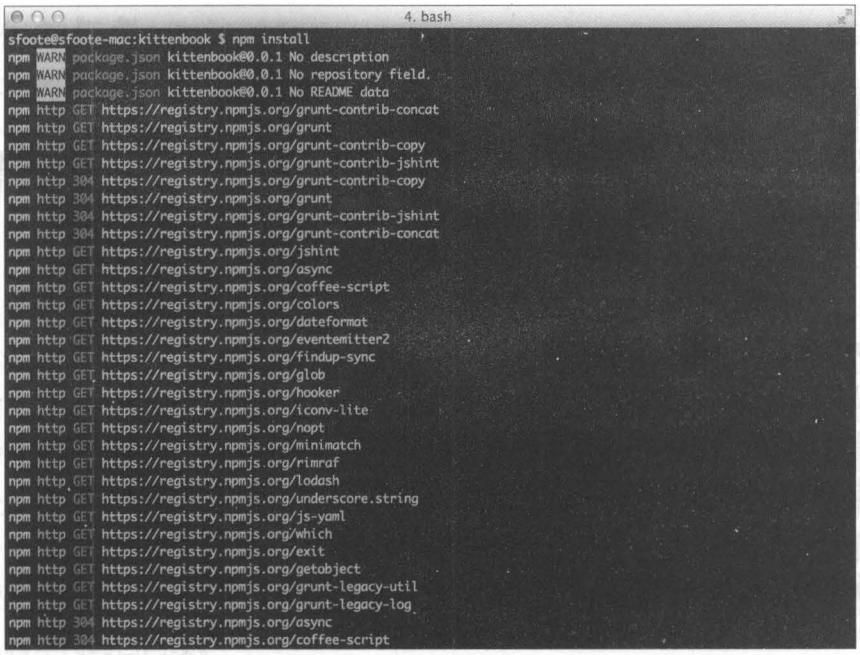

图 4.6 执行 npm install 安装定义在 package.json 中的依赖，npm 执行过程中会解析依赖

帮你创造软件的软件

理论上，构建工具并不绝对必要。构建工具自动化做的每个任务，都可以通过人工手动完成。不过，有些软件项目太大、太复杂，以至于要构建一个可执行的软件包可能需要花费一周时间来手动执行全部的必要步骤。对了，还要运气足够好，能够按照正确的顺序执行全部任务。一旦搞错一步，你可能就不得不从头再来。在这种情况下，我们可以抛出一个理论。实际上，没有构建工具，你不可能在大型软件项目中工作。

甚至在小项目中，自动化任务也可以帮你节省很多时间和避免痛苦。第 3 章中，你学到了如何使用 cat 命令链接文件。既然我们的 JavaScript 文件已经写好，为了让 main.js 工作，文件需要以正确的顺序连接。另外，每次你修改了任意一个文件，你都要再次执行 cat 命令来链接文件。这会带来很多输入工作，而输错一个字符可能会把全部都毁了。如果未来我们添加了另一个 JavaScript 文件（我们确实会），会带来更多的输入工作，以及更大的出错空间。假如在输入命令时忘记包含某个文件呢？当我们配置好 Grunt，你就可以仅仅输入 grunt，所有的文件都会被以正确的顺

序连接好。

避免错误

必须承认，我在一开始没有学习使用构建工具，直到我学了一些编程基本知识。我当时知道有种叫作构建工具的东西，但我觉得没有足够时间来学习它们。当时我注意到的工具是 Ant 和 `make`，下面这段话摘自 Ant 网站：

> Apache Ant 是一个 Java 类库和命令行工具，其任务是将处理过程描述为构建文件中相互依赖的目标和扩展点。

现在我也没搞明白这段话的意思，所以我一点儿也不觉得惊讶，当时没有决定深入学习构建工具。虽然不觉得惊讶，但我浪费了很多时间和效率。当时我在建一个网站，每次我想要做更新，就需要按照一个很长的手动任务清单执行操作。如果其中一个任务没有按照正确的方式、正确的顺序运行，我就要停掉在为所有用户提供服务的整个应用（这种事情发生的次数比我承认的还要多）。此外，我不是连接 JavaScript 文件，而是把我的代码全部写入一个巨大的、难以维护的、错误百出的文件中。因为我并不理解构建工具，我做出了很多设计错误（单一巨大的 JavaScript 文件）以及运行错误（复制错误文件到错误位置）。要是我一开始就花些时间去理解一下构建工具就好了，真希望那时候有像 Grunt 这样简单的构建工具可以用。

更快地工作

即使你有完美的注意力和完美的记忆力（比如你在执行长时间复杂任务时从不出错），你依然会因为手动操作而浪费大量时间。当软件项目变得足够大和复杂时，构建工具就是不可或缺的。如果你等到项目那么大才开始做自动化，你会发现自己处于很不利的境地。现在就花些时间学习，你会在未来节省大量的时间。

在第 9 章“编程标准”和第 15 章“高级主题”中，我们会讨论与其他人一起构建软件的来龙去脉。当你在一个软件项目中和其他人一起工作时，构建工具变得更加重要。团队中的每个成员都会工作在项目的一小部分，而每个人要测试他们那一部分时，所有其他部分都要被构建出来。没有构建工具，团队中的每个人必须要完整地理解团队里其他人做的工作，才足以手动正确地执行所有构建步骤。如果不用花时间去记其他人的构建步骤，那么你就可以做更多的事情。

自动化的任务

不是每件事都可以或者应该被自动化。你必须写代码，必须决定哪个按钮会去哪里，你还必须决定颜色、大小及文本的位置。编程的这些部分需要做大量的决策，这是构建工具不具备的特质。但很多编程任务非常适合自动化。

编译

当你使用一门编译型语言工作时，必须在每次修改代码后执行编译。由于源代码被写入一个文件，而已编译代码被写入另一个文件，当源代码文件发生了改变，则这两个文件可能会产生不同步。当这样的冲突出现时，你的源代码才是真实来源。已编译代码仅仅是源代码在某一个特定时间的快照（见图 4.7）。这些冲突的类型可能会让人困惑和抓狂（我的源代码是在做正确的事，但我的软件却并没有做那件事！）。构建工具缓解这种痛苦，它可以在每次保存源代码文件时自动执行编译，或者每次运行软件时自动编译。任何一种方式，都确保当你的软件运行时，你的源代码和已编译代码已同步。

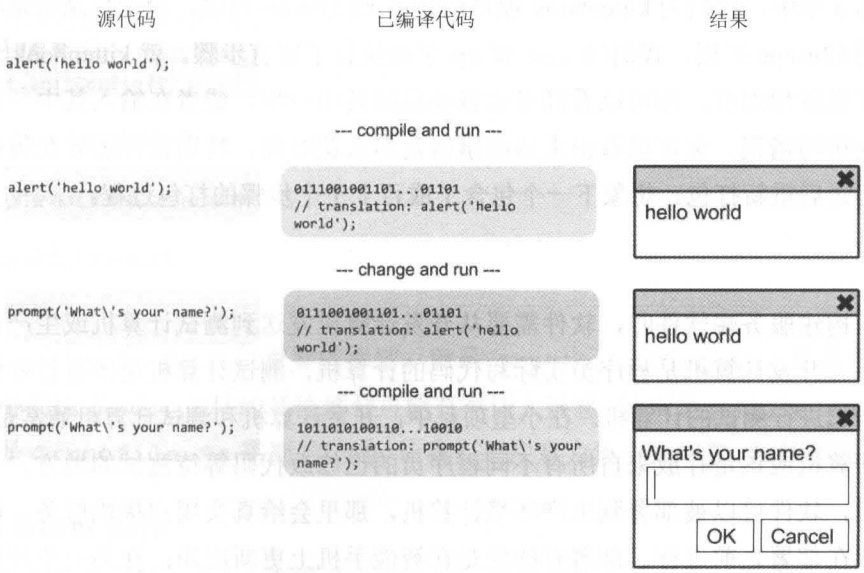

图 4.7 当源代码和已编译代码不同步时会产生困惑

测试

在第 12 章“测试和调试”中你会学习编写代码来测试你的代码。很疯狂，是吧？你可以使用构建工具来自动运行那些测试，每次修改代码，就跑一次。如果你破坏了什么东西，不需要做任何手动测试，你就能发现。如果你写了完善的测试，你会明确知道你破坏了什么。你还可以配置你的构建工具，在打包或者部署之前运行测试，如果测试失败，你就可以提前停掉它们。

打包

当编写的软件需要分发给用户的计算机并运行在其上时，软件就需要做好分发的准备，有时叫作打包。几乎所有包含超过一个文件的软件都需要进行某种形式的打包。打包过程的一部分是编译源代码，其他部分包括把文件从一个地方移动到另一个地方、连接文件、重命名文件、更新版本号，以及创建和删除文件及文件夹。就像画家不会直接在镶在画框中的画布上作画，你工作时的项目也极少与呈现给最终用户的格式一模一样。构建工具让你快速自动执行打包过程，不会出任何错误。

第 3 章中，我们为 kittenbook 项目创建了 `release` 目录，这个目录就是我们已打包的 Chrome 扩展。我们用 `cat` 和 `cp` 手动执行了所有步骤。就 kittenbook 这样简单的打包过程而言，你可以看到多么容易忘掉其中一步，或者在输入其中一条命令时出现拼写错误。你可以看出来这些事情是多么花时间，特别是你需要在每次修改源代码之后重新打包。想象下一个包含了成百上千个步骤的打包过程。

部署

当构建服务端软件时，软件需要从开发计算机发送到测试计算机或生产环境计算机上。开发计算机是程序员实际写代码的计算机，测试计算机是所有打好包的软件在一起进行测试的计算机。在小型项目中，开发计算机和测试计算机通常是一个。测试计算机应该是存放来自所有不同程序员的已修改代码等待测试的地方。一旦测试完成，软件可以被部署到生产环境计算机，那里会给真实用户提供服务。打包通常必须在部署之前进行。部署有些像是在智能手机上更新应用：在另一个计算机上开发和打包的软件更新，被用来更新已经安装在用户访问的计算机上的软件。没有构建工具，部署可能是很复杂的过程；有了构建工具，它可能就是简单到只需执行一行代码，或是按一个按钮。

构建你自己的构建过程

理论讲得足够多了。我们该实现 kittenbook 的构建工具，并从中获益了。第一步，要决定我们到底自动化什么。JavaScript 打包前不需要编译，所以我们不需要操心。我们还没有给项目创建任何测试，但我们仍然可以做一些不用自己写的测试。我们肯定要打包我们的 Chrome 扩展，处理那些连接、移动、重命名等那些我们第 3 章里用命令行干的那些事。部署扩展什么都不需要我们做，所以我们也不用管这一步。总结一下，我们要用构建工具来处理测试和打包。

Gruntfile.js

要给你的项目设置 Grunt，你需要在 kittenbook 目录下新建一个名为 Gruntfile.js 的 JavaScript 文件。Gruntfile.js 用来告诉 Grunt，哪些要自动化，以及如何自动化。Gruntfile.js 的结构是严格且清晰定义的。我们从清单 4.3 中的基本结构开始，然后填写具体信息。

清单 4.3 基本 Gruntfile.js 结构

```
module.exports = funciton(grunt){
  // 项目配置
  grunt.initConfig({
    /*
     * 我们会在这里配置任务 */
  });
  // 我们会在这里加载 Grunt 插件

  // 我们会在这里注册任务
};
```

Gruntfile.js 里最重要的事情，就是 grunt.initConfig。这是你配置每个任务的地方。这个文件的其他部分可能有点让人疑惑，特别是 module.exports，但这是 Gruntfile.js 需要的结构。让我们把注意力集中到配置、加载和注册任务。

使用 Grunt 插件

如果没有插件，Grunt 不能做任何事情。你需要插件来自动化所有的构建任务。我们会使用 grunt-contrib-concat、grunt-contrib-copy 及 grunt-contrib-jshint（Grunt 插件中，以 grunt-contrib 开头，由编写和维

护 Grunt 的那些人来编写和维护），这些我们已经加入到了 package.json 中，并使用 npm install 安装过了。现在我们需要把它们加载到 Grunt 中，并配置好，见清单 4.4。

清单 4.4　kittenbook 项目的完整 Gruntfile.js

```
module.exports = function(grunt) {
    grunt.initConfig({
      concat: {
        release: {
          src: ['js/values.js', 'js/prompt.js'], dest: 'release/main.js'
        }
      },
      copy: {
        release: {
          src: 'manifest.json',
          dest: 'release/manifest.json'
        }
      },
      jshint: {
        files: ['js/values.js', 'js/prompt.js']
      }
    });

    // 加载 Grunt 插件
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-copy');
    grunt.loadNpmTasks('grunt-contrib-jshint');

    // 注册任务
    grunt.registerTask('default', ['jshint', 'concat', 'copy']);
};
```

清单 4.4 有很多新的容易让人困惑的代码。我们来仔细看一下，确保每件事情都清晰明了。首先，我们添加了很多{和}符号（叫作大括号），和一些[和]（叫作中括号或者方括号）。在第 5 章中，你会学到很多关于大括号、中括号的知识，以及它们分别代表什么。你已经在 manifest.json 和 package.json 中看到它们。姑且认为，这些符号在 JavaScript 中是用来组织代码的。在 Gruntfile.js 中，这些结构化的代码代表了任务配置。我们声明的每个任务都有一个配置：concat:、copy:、jshint:。最后，grunt.loadNpmTasks 和 grunt.registerTask 分别用来加载插件和注册任务。

连接

在我们开始尝试自动化任务之前，最好确保我们能够用语言描述这个任务。如果你不确定你希望什么事情发生，你怎么能告诉计算机为你做什么呢？识别并澄清一个问题的过程是计划的重要部分（我们会在第 11 章“计划”中讨论》）。在我们能够理解连接任务的配置之前（见清单 4.5），我们需要明白我们到底想要什么。

1. 按顺序连接 `js/values.js` 以及 `js/promt.js` 的内容。
2. 把第一步中的输出发送至 `release/main.js`。

清单 4.5 连接任务的配置

```
concat: {
  release: {
    src: ['js/values.js', 'js/prompt.js'],
    dest: 'release/main.js'
  }
},
```

准确理解我们要执行哪些步骤，让我们更容易理解那些配置。首先，`concat:` 是配置的任务类型。其次，`release:` 是我们要执行的 cancat 任务的名字。在我们的例子中，我们只有一个 concat 任务，我们将其称为 `release`。在 `release` 任务中，我们可以看到 `src` 和 `dest`，分别代表源文件（source）和目标文件（destination）。`src` 旁边是文件名列表，这些是我们要连接的源文件，它们会按照列出来的顺序连接起来。在 `dest` 旁边是一个文件名，就是连接了源文件后输出的目标文件。`grunt-contrib-concat` 插件会在后台处理真正的连接操作，以及输出操作。

复制

现在你知道 `concat` 任务如何工作了，复制任务应该相对容易理解。首先，我们先想想需要复制任务做些什么。

复制 `manifest.json` 的内容到 `release/manifest.json`。

好吧，其实就这么一步。任务很简单，所以复制任务的配置应该也很简单，见清单 4.6。

清单 4.6　复制任务的配置

```
copy: {
  release: {
    src: 'manifest.json',
    dest: 'release/manifest.json'
  }
},
```

基于你所了解的 concat 任务，估计你可以弄明白 copy 任务在做什么。重复一遍，我们有个叫作 release 的 copy 任务，有个源文件 src 和目标文件 dest。

JSHint

JavaScript 是一门强大的语言，但是它有一些特性/bug，允许质量低下的代码依然能够工作。帮你写出更好 JavaScript 代码的一种方式，就是使用像 JSHint 这样的工具。

JSHint 会检测一个 JavaScript 文件，查找常见错误（这一过程叫作静态检查）。简单说来，它会检测文件，确保你没有不当使用这门语言，以及你没有滥用某个 bug 特性。除了帮助提升你的代码质量，JSHint 还可以教你很多关于如何写好 JavaScript 代码的知识。在很多其他语言里也有类似的工具。

jshint 任务的配置（见清单 4.7）相比 concat 和 copy 要简单得多。我们要做的全部事情就是添加一个文件列表给 files:。JSHint 会检测列表中的每个文件。grunt-contrib-jshint 的一个很棒的特性是，如果发现错误，后续的任务不会执行，错误的描述会显示在终端。

清单 4.7　jshint 任务的配置

```
jshint: {
  files: ['js/values.js', 'js/prompt.js']
}
```

加载 Grunt 插件

插件已经通过 npm install 下载并安装，这并不意味着 Grunt 知道它们的存在，只意味着我们告诉 Grunt 用它们，Grunt 就可以用。我们可以用 grunt.loadNpmTask 来达到这一目的，见清单 4.8。

清单 4.8 使用 grunt.loadNpmTasks 来加载插件

```
// 加载 Grunt 插件
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-copy');
grunt.loadNpmTasks('grunt-contrib-jshint');
```

注册任务

最后，我们需要注册一些任务来运行。这一步不是完全必要，但它是锦上添花的存在。你可能已经单独运行过 concat、copy 和 jshint 任务——打开命令行，导航到 kittenbook 目录，输入 **grunt jshint**（见图 4.8）。

这样可以工作，concat 和 copy 也没问题，但我们不想一条一条分别运行它们：这比没有 Grunt 的时候没好多少。使用 grunt.registerTask，Grunt 让你可以注册一个任务，它可以按顺序运行另外的一系列任务。我们注册一个任务，叫作 default，执行 jshint，然后 concat，然后 copy。对于 Grunt，default 是一个特别的任务名，你只要从命令行输入 grunt 就可以运行它，见清单 4.9 和图 4.9。

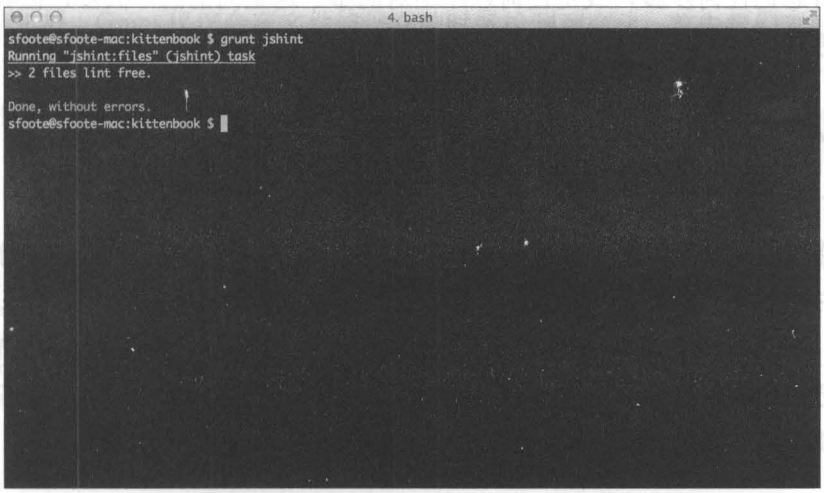

图 4.8 运行 grunt jshint，可以工作！

清单 4.9 注册新任务

```
// 注册任务
grunt.registerTask('default', ['jshint', 'concat', 'copy']);
```

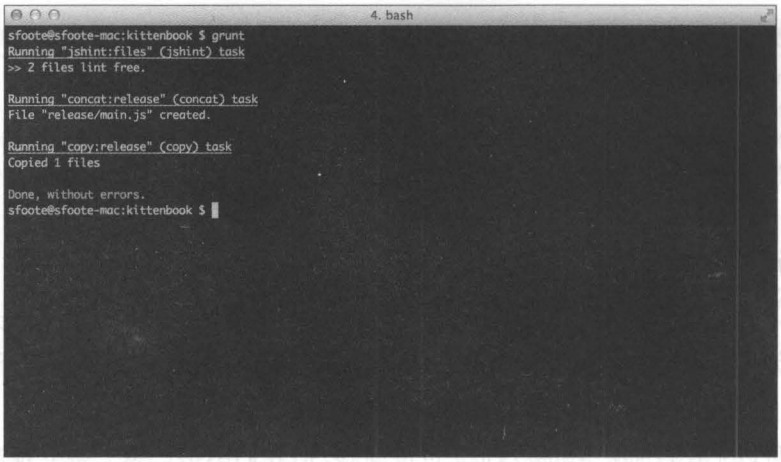

图 4.9　所有事情都配好，只要运行 grunt，所有任务都会按照正确的顺序执行

还记得 jshint 任务中，如果发现 JavaScript 代码中的错误，怎么阻止后续任务执行？我们首先运行 jshint 任务，这样一旦 JavaScript 没有达到标准，我们就可以组织这个 Chrome 扩展去打包。现在我们可以用一行命令来打包我们的扩展，并确保我们没有忘记任何步骤，或是任何顺序不对的事。这是向前的一大步，但我们还可以做得更好。

看好了

我们能不能一行命令都不用输入呢？我是说，每次修改一个文件，正式到 Chrome 上测试修改前，都要运行一次 grunt。Grunt 能不能在每次我们做一个修改前，就魔术般地打包我们的扩展呢？尽管不会什么魔术，Grunt 可以通过一个叫作 grunt-contrib-watch 的插件来实现这个目标。当我们下载并安装其他插件时，我们是把它们加入到 package.json 并运行 npm install。这一次，我们要先下载并安装，然后（自动地）新添加一行到 package.json，见清单 4.10。

清单 4.10　一行命令安装 grunt-contrib-watch，并把它加入到 package.json 中

```
sfoote@sfoote-mac:kittenbook $ npm install grunt-contrib-watch --save-dev
```

要安装插件，我们再一次在 kittenbook 目录下运行 npm install，但这次，我们加上一个包名（grunt-contrib-watch）和一个标记（--save-dev）。这个

包名是告诉 npm，我们仅仅想要安装给定的包，而这个标记告诉 npm，我们要把这个包加入到 package.json 的 devDependencies 中。去试试，在运行命令前后打开 package.json 看看。你会发现，有一行数据加进去了（见清单 4.11）。

清单 4.11 更新 devDependencies

```
{
  "name": "kittenbook",
  "version": "0.0.1",
  "devDependencies": {
      "grunt": "~0.4.2",
      "grunt-contrib-concat": "~0.3.0",
      "grunt-contrib-jshint": "~0.6.3",
      "grunt-contrib-copy": "~0.5.0",
      "grunt-contrib-watch": "~0.5.3"
  }
}
```

现在我们可以在 Gruntfile.js 里配置一个监控任务。我们要 Grunt 来监控 manifest.json 及所有 JavaScript 文件的变化，然后每次其中一个文件变化时就打包我们的扩展。Grunt 仅仅会在文件被保存的时候检测到变化。首先我们需要配置这个 watch 任务（见清单 4.12）。这个 watch 配置包括了一个文件清单（就像 jshint）和一个任务清单。一旦 files 清单中的一个文件发生变化，tasks 清单中的任务就会执行。

清单 4.12 添加 watch 配置到 grunt.initConfig

```
grunt.initConfig({
    concat: {
      release: {
        src: ['js/values.js', 'js/prompt.js'],
        dest: 'release/main.js'
      }
    },
    copy: {
      release: {
        src: 'manifest.json',
        dest: 'release/manifest.json'
      }
    },
    jshint: {
      files:['js/values.js', 'js/prompt.js']
```

```
    },
    watch: {
      files:['<%= jshint.files %>', 'manifest.json'],
      tasks: ['default']
    }
});

// 加载 Grunt 插件
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-copy');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-watch');
```

因为我们在 jshint 文件清单中列出了全部的 JavaScript 文件，我们不需要再在 watch 文件清单中列出它们。取而代之，我们可以使用一个特殊的 Grunt 专用语法：<%= jshint.files %>，来引用相同的清单。我们还想要在 manifest.json 变化时打包扩展，所以把它也加入到清单中。我们要执行的任务只有 default，它会按照正确的顺序运行另外三个打包任务。试试看，在命令行中输入 grunt watch，然后修改一个 JavaScript 文件，见图 4.10。

Grunt 的 watch 任务会保持运行，直到你让它停下来，而且你可能会注意到那里没有关闭或取消按钮可以按。你可以通过按下 Ctrl+C 组合键来停止 grunt watch（以及几乎其他任意命令行命令）。或者，你可以关掉终端窗口，然后任何正在执行的任务都会被杀掉。

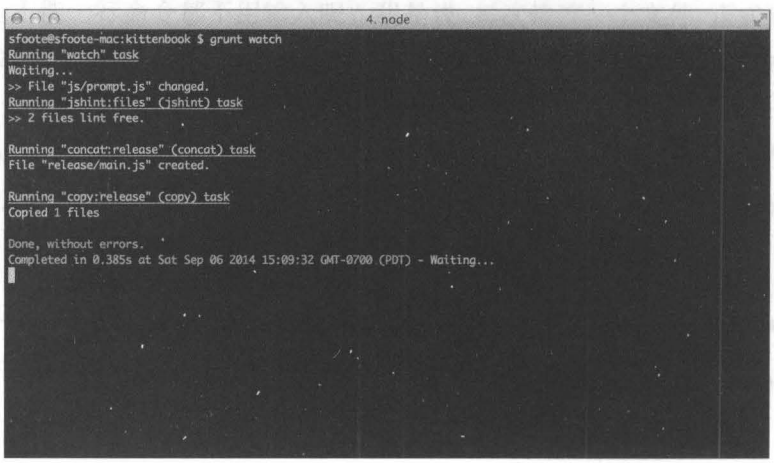

图 4.10　运行 grunt watch 一次，你的扩展会在每次做出修改时自动被打包

在本章，你完成了一些惊人的工作。你安装了构建工具，并给你的扩展配置了每次修改自动打包功能。你还给未来添加新的任务打好了基础。在本书的剩余部分，我们会给 `Gruntfile.js` 文件添加和修改内容，本章你做的工作是必不可少的一部分。做得好！

总结

本章你学到了：

- 如何避免错误，并使用构建工具工作得更快
- 什么类型的编程任务适合自动化
- 如何通过安装 `node.js` 和 Grunt 来配置你自己的构建工具
- 如何给你的项目配置 Grunt

坦白地说，大部分程序员直到以错误的方式工作了很久（包括我）之后，才会开始学习使用构建工具。本章中你学到的概念和技巧会让你成为一个更好的、更有价值的程序员，不论你是一个人工作还是在团队中工作。

下一章中，你会学到很多关于数据的知识，具体来说：

- 不同的数据类型
- 你的 JavaScript 和 JSON 文件中的那些{和[符号是什么意思
- 如何组织数据，以及为什么它很重要
- 数据库是什么，以及如何工作

5

数据（类型）、数据（结构）、数据（库）

注

项目：使用 JavaScript 数组存储 Facebook 订阅中的图片数据。

作为一个高效的程序员，你要了解数据，除了是数据越多越好，大数据还可以洞察有趣的事情。你需要知道不同类型的数据，以及如何正确使用它们。你需要知道如何存储和取出数据。几乎每个计算机程序都要以某种方式来处理数据。在构建 kittenbook 扩展时，你已经处理过一些不同类型的数据，而且 `Gruntfile.js` 也有一些其他类型。你可能现在还没理解“{”和“[”这类奇怪符号所代表的不同数据类型。我们马上改变这种状态。

数据类型

有很多不同类型的数据，有些类型由其他类型组成。你已经知道的那些数据类型：

- 数值类型
- 字符类型
- 字符串类型
- 时间类型

其他你可能熟悉的：

- 布尔类型

- 集合类型
- 数组类型

以及另外一些你可能从没听说过的：

- 浮点类型
- 长整型
- 哈希类型

为什么存在不同的数据类型

存在不同数据类型，是因为计算机必须以不同的方式存储和处理不同类型的数据。计算机在存储数字 4 时预留的空间要比存储一个长字符串小很多（这本书可能是一个大字符串）。有时不同类型的数据无法共存，举个例子，如清单 5.1 所示，你可以将数字相加得到数字，也可以将字符串相加得到字符串，你还可以将数字和字符串相加，得到……字符串？将 2 加上'2'时得到'22'是有点荒谬，但得到 4 就更合理么？字符串'2'和数字 2 之间的区别对于计算机来说是非常大的，混用不同类型会导致你的程序得到奇怪和意外的结果。事实上，在一些语言中，尝试将一个数字和一个字符串相加，会给你提示一个错误（参见本章中关于静态和动态类型语言的小节）。

清单 5.1　混用数据类型

```
var numbers = 2 + 2;        // 4
var strings = '2' + '2';     // '22'
var numberStrings = 2 + '2'; // '22'
```

基本数据类型

基本数据类型是指单块数据（例如一个数字，或者一个字符）。基本数据类型通常叫作原子类型，因为它们不能被进一步拆分。

布尔类型

可能最重要的数据类型就是布尔类型。布尔类型（以英国数学家乔治·布尔命名）是最简单的数据类型：1 或 0，开或关，是或否，真或假。计算机中所有的逻辑都依赖于布尔数据类型。在第 7 章“何时使用 IF, For, While”中，你会学到关于

if 语句和 while 语句的知识，以及如何使用布尔类型来控制你的计算机如何工作。试一下清单 5.2 中的代码，看看布尔类型如何发挥作用。关于如何运行代码，可以看一下侧边栏。

清单 5.2 真或假

```
if (true) {
  // console.log 向控制台打印一条消息
  // 就像 alert，但是不会弹出烦人的窗口
  console.log('It is true!');
}
if (false) {
  console.log('This line will never be executed.');
}
```

使用 Chrome 浏览器的开发者工具做实验

我喜欢 Web 编程。你可以看到自己访问的每个网站的源代码，只要你知道从哪看。这太令我激动了，因为那意味着，我可以通过看其他网站如何解决类似问题，来解决自己的问题。这一暗藏的知识池，是 Web 技术飞速发展的原因之一。如果你正尝试解决一个问题，并且你找到了一个 Web 页面，已经解决了这个问题，你就可以看看源代码，看它是如何做的。当然，寻找灵感和复制粘贴他人的代码有区别。前者可接受，也是 Web 的重要组成部分；后者在未经许可时不允许。

这些全部力量的关键，是在哪看源代码。大部分浏览器都有开发人员可以使用的工具，用来查看源代码。IE 有开发者工具（按 F12 可以看到），FireFox 有 Firefox 开发者工具，Chrome 也有开发工具。我们在写 Chrome 扩展，所以聚焦在 Chrome 开发工具上，但如果你在构建网站，那应该能使用任意浏览器的开发者工具。

要打开 Chrome 开发工具，可以右键点击任意网页，然后点击菜单中的“审查元素”（见图 5.1）；或者可以使用键盘快捷键（我的最爱）：在 Windows 和 Linux 上用 Ctrl+Shift+I 组合键，或者在 Mac 上用 Command+Option+I 组合键。或者，点击 Chrome 浏览器菜单按钮，选择“工具”→“开发者工具”（见图 5.2）。你可以看到一个窗口出现在网页的底部，可能会在左侧看到一些 HTML 代码（见图 5.3）。这个窗口中有很多内容，第 12 章“测试和调试”中的“调试”一节中将介绍更多。现在，我们只是要用到 Console 标签。

你可以使用 Chrome 开发工具查看你正在浏览的网站源代码，但当前，我们只

要使用 Chrome 开发工具来快速尝试代码，试验新概念，所以我们打开哪个网站并不重要。还记得上一章学到的 Node.js 的交互模式吗？我们可以在 Chrome 开发工具的控制台做类似的事情（见图 5.4）。你可以按下 Esc 键打开控制台，或者点击上边栏的“控制台”按钮。现在输入清单 5.2 中的代码。

提示：每次按下回车键，不管你之前输入了什么，都会被执行。想要不执行代码添加一个新的行，按下 Shift+Enter 组合键。

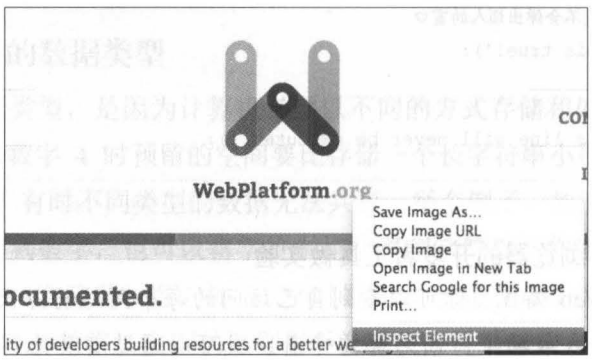

图 5.1　点击“审查元素”打开 Chrome 开发工具

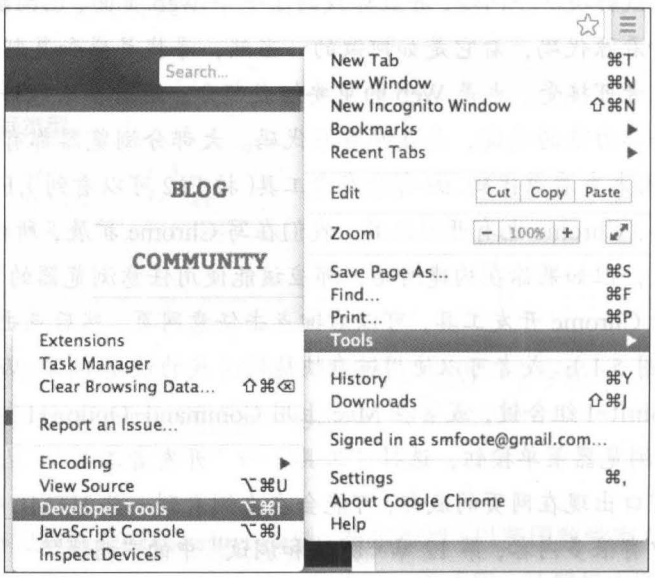

图 5.2　使用 Chrome 菜单打开 Chrome 开发工具

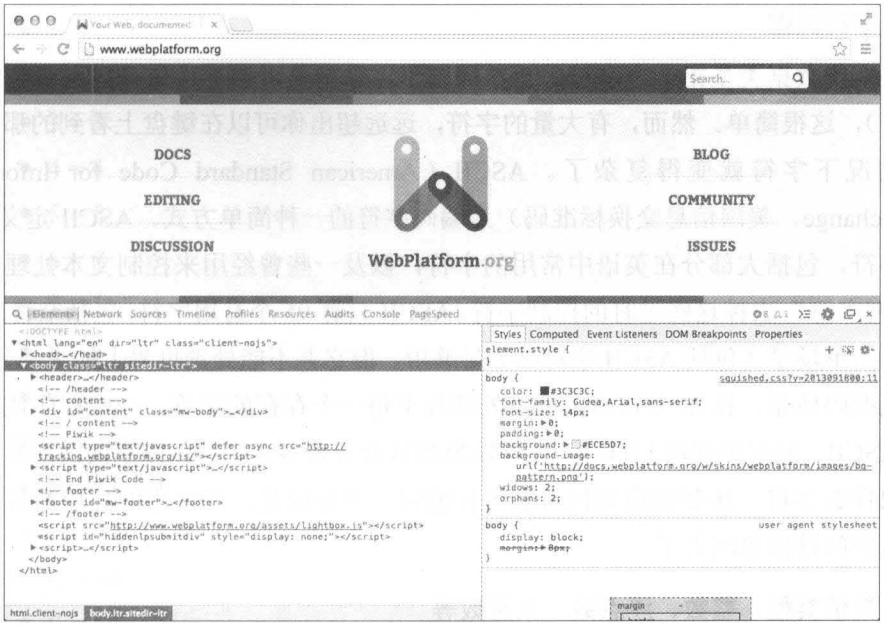

图 5.3 初次打开 Chrome 开发工具的页面，很快，你就会感激它的优美

图 5.4 在 Chrome 开发工具中运行清单 5.2 的代码

字符类型

字符数据类型指一个字符。正常情况下，是指英语字符（在键盘上看到的那些字符），这很简单。然而，有大量的字符，远远超出你可以在键盘上看到的那些，那些情况下字符就变得复杂了。ASCII（American Standard Code for Information Interchange，美国信息交换标准码）是编码字符的一种简单方式。ASCII 定义了 128 个字符，包括大部分在英语中常用的字符，以及一些曾经用来控制文本处理的过时“控制字符”。去掉这些过时的控制字符，ASCII 只有 95 个有用字符，包括空格。ASCII 对于简单场景（包括 ASCII 字符画）足够用，但它并不能处理世界上的大部分语言。其他编码体系，比如 UTF-8，可以处理几乎每一个存在的字符。一些程序使用简单的 ASCII，但很多使用 UTF-8。你可以想想这会带来多少困惑：你不仅仅需要知道你要用什么字符，还要知道如何对它进行编码。幸运的是，你使用的环境通常都为你把这个问题抽象出去了。

数值类型：整数、浮点数、长整数等

数字可以很小，几乎不需要多大空间来存储；也可能非常非常大，需要大量的空间来存储。我们并不假设所有的数字都非常大，导致存储它们会浪费很多空间；我们也并不假设所有数字都很小，导致没有足够的空间来存储非常大的数字。你会发现，有很多不同类型的数字，有的占用空间多些，有的占用空间少些。

从编程角度来说，短整数是那些没有点的小数字。可以作为整数存储的范围从 -32000 到 32000。如果你需要这个范围之外的数字，则要将它存储为一个长整型。长整型和短整形类似，但具有更大的范围（几十亿或是更大，视语言而定）。如果你需要存储带有小数点的数字，你需要使用浮点型。浮点型的名字来源于小数点在一个数字中间的某处浮动。在某些语言中，你必须显式地声明你要用哪种类型的数值，不过 JavaScript 会自动找到最适合当前场景的数值类型。

16 进制是什么？

你现在知道了不同类型的数值，但你可能还没意识到你有很多种存储数字的方式。大多数时候，你会使用十进制数字（这是你习惯的格式），但有时你需要能够使用其他类型，比如二进制数或十六进制数。你最可能使用十六进制数的场景之一就是定义颜色（特别是在网页中）。十六进制数使用 0～9 和 A～F：

- 0 = 0

- 1 = 1
- 2 = 2
- 3 = 3
- 4 = 4
- 5 = 5
- 6 = 6
- 7 = 7
- 8 = 8
- 9 = 9
- A = 10
- B = 11
- C = 12
- D = 13
- E = 14
- F = 15
- 10 = 16
- 11 = 17
- …
- FF = 255

这些可能看起来有点让人迷惑，但你可以找到一些工具帮你将十进制数转换成十六进制数，所以你不用知道如何转换。我提到这个话题，是因为你可能在一些应该是数字的地方看到字母；当你看到字母，你就知道那可能是十六进制数。

组合数据类型

有些数据类型由其他类型的数据组合而成。它们可能用基本数据类型组成，也可能是其他组合数据类型，或者两者皆有。

字符串

字符串可能是最简单的组合数据类型，它就是字符的组合。我觉得字符串很重要，因为它们可以算是最容易理解的数据类型。基本上说，它们就是文本，而大部分人都能够理解文本的概念。字符串在计算机程序中到处都会用到。你已经在

prompt.js 中以两种方式用到了字符串。第一种，将用户的名字以字符串的方式存储到变量 userName 中。第二种，通过连接一些字符串（包括存储在 values.js 中的变量）构造一个消息显示给用户。当我意识到，我在计算机上看到的每一条消息都是一个人输入进去的，那对我来说就像是一个“神启”。

然而字符串还可以用来做显示信息以外的事情。事实上，源代码文件就是一个大字符串。当源代码被发送给编译器或者解释器时，它作为字符串发送。编译器的工作就是解决这个字符串代表的意义。字符串还可以用于在程序之间发送消息，定位资源，以及提供用户认证识别号。尽管用户 ID 大部分是数字，但它们通常以字符串的方式存储。你不能在字符串上执行数学运算（加减乘除等），而且用户 ID 用于识别用户，你不会想要在它上面做数学运算。

正则表达式

正则表达式是一个工具，用来识别字符串中的模式。一个正则表达式，看起来就像字符串，但每个字符都有其意义。正则表达式就像一个隐秘的微型编程语言，居于其他编程语言之中。正则表达式是一类有趣的数据，因为它们是能够做事情的数据。

在大部分语言中，正则表达式都是处于两个斜杠之间的一组字符：/regular expression/。我们有一整章（第 6 章“正则表达式”）关于正则表达式的内容，在这就不多讲了。记住一点：正则表达式可能会很难理解，但是它们也可能非常强大。

时间

很多常见的编程语言（包括 JavaScript，Java，Perl，Python，Ruby，以及 C）使用一个叫作 Unix 时间的系统来跟踪时间。在 Unix 时间中，一个时间用自“世纪之初”开始所经过的秒数来表示。这个“世纪”始于协调世界时间（UTC）1970 年 1 月 1 日 0 点 0 分 0 秒。那一时刻在 Unix 时间中表示为 0。自那一刻起，每过一秒，这个数字就增加 1。

在很多语言中，一个“时间”不仅仅是一个有特殊意义的数字，相反，一个时间被存储为一个带有特定的属性和方法对象（见字典，也称为映射、哈希、对象及关联数组），你可以操作那个数字。你已经在 values.js 中见过了，我们用 new Date 创建了一个时间对象，然后我们在 Date 对象上使用类似 getFullYear 的方法，以获取关于那个时间的人类可读信息。

又一个千禧年危机（Y2K）

Unix 时间存储为 32 位有符号整数（一个有符号整数可以是正的，也可以是负的），你之前已经学过，一个数字不可能大过给定的范围。一个有符号 32 位整数最大是 2^{31}，或者 2,147,483,648，并且如果你要再加 1，这个数字就会反转成 -2,147,483,648（32 位可表示的最小数字）。UTC 时间 2038 年 1 月 19 日，凌晨 3:14:07，将会是自“世纪之初”后的第 2,147,483,648 秒。下一秒，也就是 2038 年 1 月 19 日，凌晨 3:14:08，所有使用 Unix 时间的计算机系统就会认为当前时间是 1901 年的 12 月 13 日。这将会产生问题。一个解决方案是使用 64 位有符号整数来替代 32 位数字，使用 64 位数字仍然会有上下极限值，但这个极限值是目前所估计宇宙寿命的 20 倍，大概是从现在开始的 2,920 亿年后。我觉得我们可以暂时接受这个值。

数组

数组是一些东西的有序列表。数组的概念几乎在每一个编程语言中都存在，但它们的名字，以及工作原理可能会有很大不同。当你在某些编程语言中创建数组时，你必须声明数组保存的数据类型及数量。使用这一信息，计算机可以在内存中找到一段连续的空间来存储这个数组。其他语言（包括 JavaScript）就更随意一些，你不用提前知道数组中所存储东西的内容和数量，因为数组可以保存任意数据类型。

你可以把数组想象成一排有数字的盒子，数字从 0 开始（见图 5.5）。（在编程领域，数字几乎都从 0 开始。一开始我觉得这真是太奇怪了，现在已经习惯。）严格的语言会要求你声明多少个盒子，以及每个盒子要多大（例如，我需要一个足以放下字符串的盒子）。这让计算机去内存中找一个地方，适合把全部盒子排成一排放在一起。更灵活、随意的语言也有带数字的盒子，但盒子可能不排成一排放在一起（见图 5.6）。严格的版本会允许更快速的查找，因为数组的项都按顺序组织到一起。灵活的版本要慢一点，因为它需要更长的时间来找到对应的盒子。

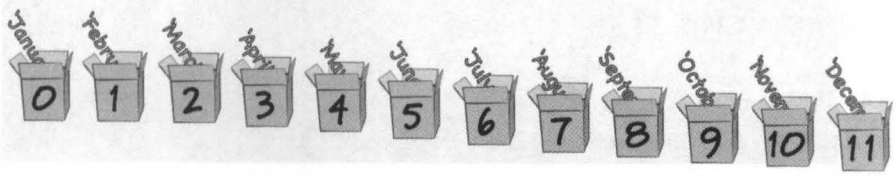

图 5.5 数组就像是一长列有数字的盒子

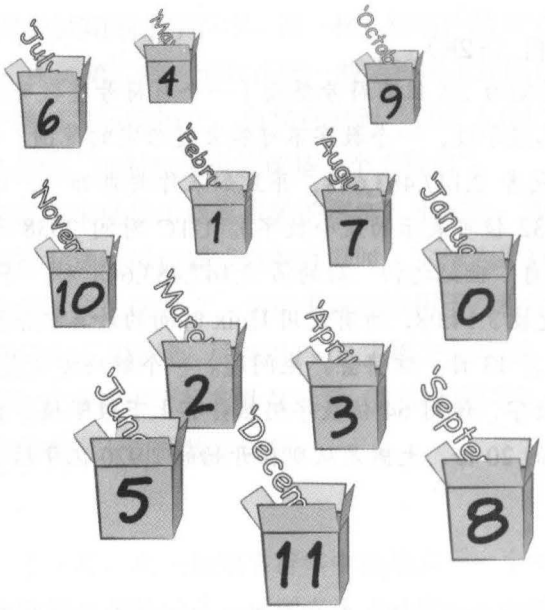

图 5.6　当数组不是严格定义时，找到对应的盒子可能需要更长的时间

我们用数组来做些实验。你可以打开 Chrome 开发工具控制台或在命令行打开 Node.js 的交互模式。首先，我们创建一个月份名字的数组。在 JavaScript 中，创建数组的最佳方式就是使用一个以逗号隔开的列表，并用一对方括号括起来（见清单 5.3）。

清单 5.3　月份名字的数组

```
var months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
              'September', 'October', 'November', 'December'];
```

你可以通过使用那个项的索引（索引就像盒子上的数字）来访问数组中的一个项。记住索引是从 0 开始，所以如果你要找数组中的第三项，其实对应的索引是 2。在 JavaScript 中，访问数组的正确语法是使用数组的变量名，然后是方括号括起来的索引，见清单 5.4 和图 5.7。

清单 5.4　访问 months 数组中的项

```
months[11];    // 这里会显示 'December' (不是 'November')
```

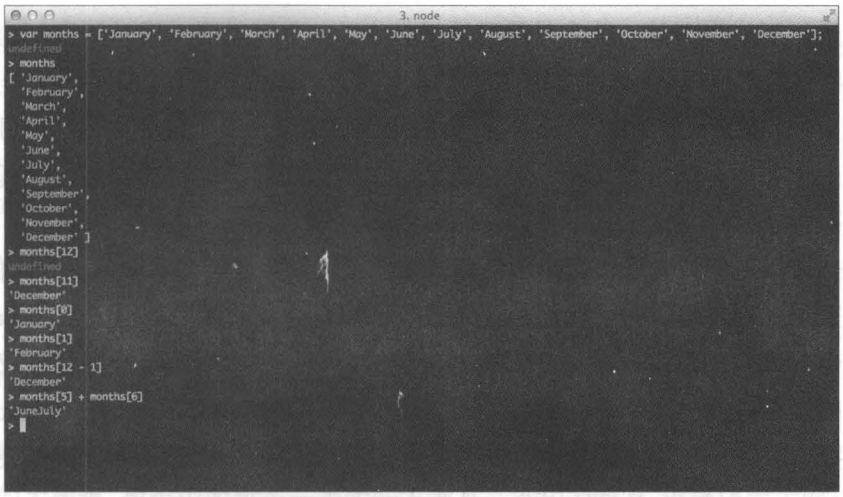

图 5.7 用交互式 Node.js 来做数组实验

像 JavaScript 这样的语言，数组可以是任意大小，数组的元素可以是任意类型，通常会有一些方法来给数组添加和移除元素。大多数情况下，你会从数组的末端添加或移除元素。在 JavaScript 中，可以分别通过 `push` 和 `pop` 来实现。运行 `push` 会添加一个元素到数组中，然后在新元素添加成功后返回给你数组的长度。运行 `pop` 会移除并返回数组的最后一个元素（见图 5.8）。你还可以用 `length`（`length` 是一个属性，不是方法，因此它不需要被调用，所以你不需要加上 `()`）来检查数组的长度。

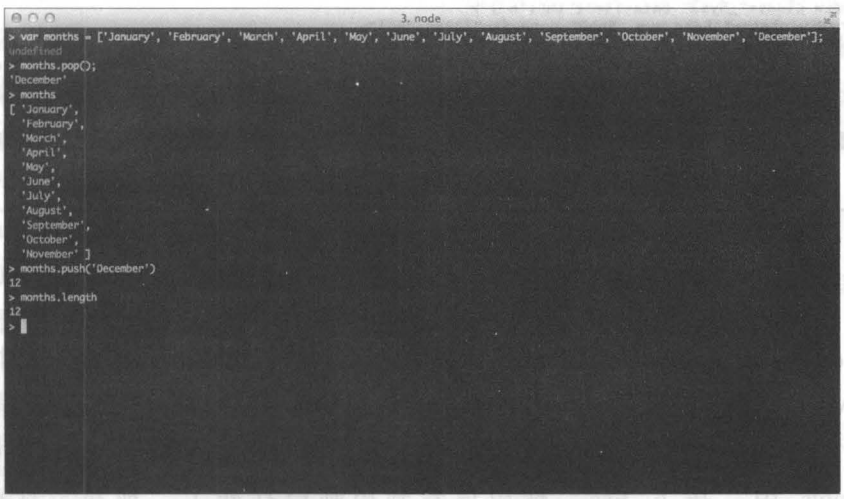

图 5.8 用 push 和 pop 从数组的尾部添加和移除元素

在进行下一步之前，花点时间用数组做实验。你可以用一周时间来创建一个数组，或者你的家庭成员，或者你最喜欢的电影列表。试着 pop（弹出）和 push（压入）一些元素。如果你从一个元素数目为 0 的数组中弹出元素，将会如何？你可以创建一个包含字符串、数字和日期的数组吗？这类实验在你学编程的新概念时真的很重要。没有尝试用过，你永远不会真正理解它。

当你可以迭代或者遍历数组中的每一个元素（你会在第 7 章中学到怎么做）时，数组就变得特别有用。对于我们的 kittenbook 项目，我们使用数组来保存 Facebook 页面上全部照片的索引，然后我们会遍历整个数组，并改变每一个图片。我们现在可以开始这个过程，创建一个数组，包含 Facebook 页面上全部照片。在 Facebook.com 的新鲜事中右键点击一张图片，点击“审查元素”显示 Chrome 开发工具的窗口（见图 5.9）。

我们不想替换全部图片，只是替换在新鲜事中的图片，所以我们需要找到在 HTML 中识别一张在新鲜事中图片的内容。在图 5.9 中，你能看到 HTML 的顶部那行：

```
<div class="clearfix userContentWrapper _5pcr">
```

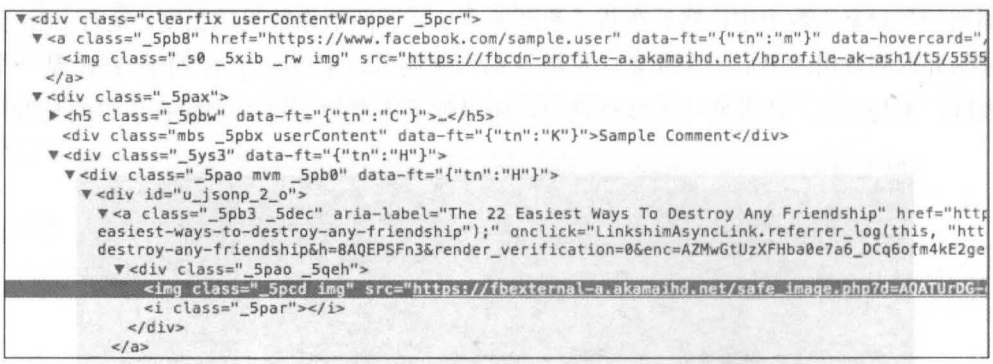

图 5.9　Chrome 开发工具显示的 Facebook.com 的 HTML

它表示我们要加入到数组中的每一张图片，都可以在一个有 `userContentWrapper` 类的 `div` 标签中找到。（注：Facebook 可能会随时改变它的 HTML。如果你没看到 `userContentWrapper`，找找其他能够识别新鲜事中图片的类）。我们用一行命令就可以创建数组，使用所谓的 `querySelector`。在 Chrome Dev Tools 中，按下 Esc，你应该看到控制台打开了。现在，你可以用

document.querySelectorAll，在所有标有 userContentWrapper 类的 div 中，找到图片（img），见清单 5.5。

清单 5.5 使用 document.querySeelctorAll 创建一个图片数组

```
var images = document.querySelectorAll('div.userContentWrapper img');
```

> **注**
>
> document.querySelectorAll 返回一个 NodeList，它很像一个数组，但 NodeList 没有 push 和 pop 方法。

在 Chrome 开发工具中尝试清单 5.5 中的代码。新变量 images 现在应该包含很多图片的大数组。数组中的每个元素实际上都是一个用 HTML 来描述真正图片的“对象”。我们后面会需要那个数组，所以先继续把清单 5.5 中的代码添加到 js 目录下的新文件中，命名为 getImages.js。现在更新 Gruntfile.js，让 Grunt 构建扩展时包含这个新文件。

字典（也称为映射 map、哈希 hash、对象、关联数组）

字典有很多名字，但我认为“字典（Dictionary）”这个名字是其中最贴切的一个。字典是一组值，其中的每一个值都有唯一的标识（这个概念通常被称为键-值对）。韦氏词典就是一组值（单词定义），其中每个值都有个标识（单词）。字典的其他名字（对象、映射、哈希、关联数组）中，有些会在你进一步熟悉其他计算机概念时，看起来更有道理。在 JavaScript 中，它们叫作对象，所以我们大部分时间会用这个名字，但了解其他称呼也很重要。

在构建 kittenbook 时，你已经用到了一些对象。还记得 JSON 就是 JavaScript *Object* Notation（JavaScript 对象标记）吧，所以 manifest.json 和 package.json 实际上都是 JavaScript 对象。你还在 Gruntfile.js 中用到了对象。JavaScript 对象使用一个花括号{，然后是一些键-值对，以及另一个花括号}来定义，见清单 5.6。

清单 5.6 一个示例 JavaScript 对象（描述的是我）

```
var author = {
  firstName: 'Steven',
  lastName: 'Foot',
  age: 27,
```

```
  favoriteFoods: ['waffles', 'Thai curry']
};
```

打开一个 JavaScript 控制台（Node.js 或者 Chrome）并尝试创建一个用于描述的 reader 对象。写 JavaScript 对象时，你需要在各个键值对之间放一个逗号，但不要在最后一对后面放逗号。

现在已经创建了对象，你会想要从对象中取到相应的值。要取出一个对象的值，可以使用对象名，然后是一个点，然后是键的名。比如，author.firstName 返回'Steven'。你还可以用这个语法修改属性值，以及创建新的属性。例如，要修改 author.lastName 的拼写错误，并添加'grilled chicken'到 favoriteFoods 数组中，我可以用清单 5.7 中的代码。一个对象中的元素，没有按照任何特定顺序存储；怎么能让元素在未来更快被找到，计算机就会怎么存储它们。不像数组，对象中的元素不能通过数字索进行引访问。

清单 5.7　修改对象的属性

```
author.lastName = 'Foote';
author.favoriteFoods.push('grilled chicken');
```

现在我们把对象放到 kittenbook 中为我们工作。你的任务是在 value.js 中创建一个对象，它包含在 prompt.js 中要用的全部值。迄今为止，我已经带你仔细演练了该如何写代码。从现在开始，我要省略一些事情，来让你自己想出细节。我要省略的第一件事，就是如何在 value.js 中创建对象。你还需要修改 prompt.js 来使用这个对象，所以它应该看起来像清单 5.8 中的样子。别忘了在命令行运行 grunt watch，这样你的修改就会在保存的时候被检测到。

清单 5.8　prompt.js 使用 pbValues 对象

```
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = '<h1>Hello, ' + userName + '!</h1>' +
                          '<p>' + kbValues.projectName + ' ' + kbValues.versionNumber +
                          ' viewed on: ' + kbValues.currentTime + '</p>';
```

动态和静态类型语言

从你写过的 JavaScript 可以看出，当创建一个新变量时，你不需要告诉 JavaScript 那个变量要存储什么数据类型。JavaScript 会弄清楚的，因为 JavaScript 是一个动态

类型语言；数据类型在程序运行过程中会被动态分配。在你写代码时，你不需要考虑用什么数据类型，因为运行时会帮你搞清楚。先行动，再提问。这一西部荒野式的做法可以让你做一些疯狂的事情，比如把一个字符串和一个数字相加，得到另一个字符串，见清单 5.1。

反过来，静态类型语言会要求你在声明新变量时声明其数据类型。静态类型语言有很多好处，你总是能知道你在处理的数据是什么类型，所以你不会遇到像 `2+'2'='22'` 这样的惊喜，因为你甚至不被允许将不同类型的变量相加。坦白说，一开始我觉得静态类型语言很难理解。我并没有真正理解用静态类型语言的好处在哪。我看到的就是更加让人迷惑的语法，需要打更多的字。例如，清单 5.9 和 5.10 对比了用 Java（静态类型）和 JavaScript（动态类型）创建字符串。对于我来说，清单 5.9 更难理解。

清单 5.9 用 Java 创建字符串

```
String greeting = new String("Hello, world!");
```

清单 5.10 用 JavaScript 创建字符串

```
var greeting = 'Hello, world!';
```

动态和静态类型，只是完成相同工作的不同哲学。动态类型给你快速工作的自由，不用考虑使用的数据类型，因为运行时会帮你搞定。静态类型给你的是一种结构，帮你避免很多在动态类型语言中常见的奇怪、痛苦的缺陷，这些缺陷都难以追踪。每一种语言都有其存在的意义，理解其中的动机，会帮助你更好地决定什么时候该用哪一个。

数据结构

软件中的数据通常用来对真实世界中的某些方面进行建模，而任何模型，都不能完美展现真实的世界。想想你刚刚创建的 `reader` 对象：它可能包含某些标签是准确描述你的，但一定还有一些它没提到而你具备的方面（我知道我除了名字，年龄和最爱的食物外还有很多）。在软件中展现你（或任何东西）的全部既不可能也不现实。你必须决定对于软件的设计目的，哪些是重要的，那些是可以忽略的；然后

你必须组织模型的重要部分，来确保正确信息容易被获取，并且处在合理的位置。

为了解释这一点，我们考虑采用三种不同的方式来组织 author 的数据。第一种选择用一个单独的字符串，包含全部值（见清单 5.11）。第二种选择用没有键只有值的数组（见清单 5.13）。第三种选择用包含键和值的对象（见前文清单 5.6）。

清单 5.11　作为字符串的 author 数据

```
var author = 'firstName=Steven&lastName=Foote&age=27&favoriteFoods=waffles,Thai curry';
```

可以看到清单 5.6 中的所有 author 数据都在清单 5.11 中表示出来，但很难读。字符串包含以&符号分割的键值对。键在=的左侧，值在右侧。数组类型的值以逗号分割。没法从字符串中区分数字或是数组。添加一个新的键值对很困难，修改已有的值甚至更难。以下是清单 5.12 中修改 age 标签的全部所需代码。

清单 5.12

```
// 在字符串中找到'age'键出现的地方
var ageLocation = author.indexOf('age=');

// 找到'age'键后面的'&'
var nextAmpersand = author.indexOf('&', ageLocation);

// 取出当前的年龄(age)，就是从 ageLocation
// 开始的 4 个字符，在 nextAmpersand 结束
var currentAge = author.substring(ageLocation + 4, nextAmpersand);

// 将 currentAge 转换成整数（使用十进制）
currentAge = parseInt(currentAge, 10);

// 给这个值加 1，因为今天是我的生日
currentAge = currentAge + 1;

// 将新的 currentAge 值放回作者（author）字符串，按照以下顺序：
// 1. 在旧的年龄值之前的字符
// 2. 新年龄
// 3. 在年龄之后，从'&'符号开始到字符串最后的字符
author = author.substring(0, ageLocation + 4) +
         currentAge +
         author.substring(nextAmpersand);
```

尽管有这么多缺点，在字符串中存储键值对实际上非常普遍。看看图 5.10 中的 URL：在?之后的部分就在使用几乎和清单 5.11 中一样的格式来组织键值对。URL

一直使用字符串来存储键值对，原因之一是字符串非常容易移植（一个编程语言创建的字符串可以很容易地在其他编程语言中使用），而其他数据结构的一致性要差很多（JavaScript 中创建的对象不能直接用在其他任何编程语言中）。

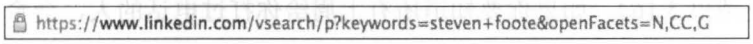

图 5.10 URL 是通常包含键值对的字符串

用数组作为数据结构（见清单 5.14），我们解决了使用字符串的主要问题，但是引发了新问题。字符串、数字、数组及其他数据类型键的区别在使用数组时很清楚，并且，添加和修改值时很直接。在清单 5.14 中，我们做了和清单 5.12 一样的事情（把 age 增加 1），但我们用一行代码就做到了。

清单 5.13 数组结构的 author 数据

```
var author = ['Steven', 'Foote', 27, ['waffles', 'Thai curry']];
```

清单 5.14 author 数据以数组存储时修改 age

```
author[2] = author[2] + 1;
```

清单 5.14 中的代码比清单 5.12 中的代码简单得多，因为数组形式的数据结构相比字符串形式的数据结构更容易读取和修改。但我们确实有新的问题。2 代表什么？我知道索引为 2 的元素描述了我的年龄，因为我刚刚写了这个数组，但是没有任何东西表明 2 跟年龄有什么关系。数组对于一列元素（例如一列我最喜欢的食物清单）很有用，但在描述什么东西时没什么用。对象在这方面要有用得多。

如清单 5.15 所示，修改以对象来组织数据的 author 的 age 标签，只需要一行代码，就和数组一样；然而，因为我们使用标签的名字，而不是索引，对象代码要更容易理解，也更不容易出错。

清单 5.15 以对象存储 author 数据时更新 age 的值

```
author.age = author.age + 1;
```

字符串、对象，以及数组可以一起用于创建更加复杂的数据结构。尽管偶尔创建全新的数据结构没问题，但通常还是使用已有的模式，比如集合、栈、树和图等，要更好。

集合

你可能已经很熟悉集合的数学概念。我以前并不知道我意识到了集合，直到我发现韦恩图就是用集合组成的，而我很熟悉韦恩图。集合中基本上不允许重复的数组存在（见清单 5.16）。如果你要知道所有上周给你打过电话的人，会希望你的手机以集合的方式展示给你。你并不介意某人给你（重复）打了多少次，你只想知道他们到底是否打给你过。很多编程语言中（尽管 JavaScript 还不是），集合是内建的组合数据类型。数学中有一整个分支关于集合及其应用的理论，如果你遇到一个需要移除重复列表的编程问题，读一下集合论会对你很有帮助。

清单 5.16　集合和数组对比

```
// 调用历史，以存在重复的数组形式保存
var callsArray = ['George', 'George', 'Elaine', 'George', 'Newman', 'George', 'George',
'George', 'Elaine', 'Kramer', 'George'];

// 相同的调用历史，以集合形式保存，消除了重复
var callsSet = ['George', 'Elaine', 'Newman', 'Kramer'];
```

栈

栈是一个数组，仅让你添加、移除，或者查看顶端的一个元素。考虑一个装着纸的栈，你可以在栈的顶端放一张纸，或从栈的顶部抽取一张纸，但如果你尝试从中间添加或抽取，事情就变得很乱。栈在保存历史记录时很有用，例如，浏览器历史的原理就像栈。在你加载一个新的网站时，网站会加到栈顶；当你点了后退按钮，栈顶的页面会被移除，前一个页面就会加载。清单 5.17 展示了浏览器历史在 JavaScript 中会怎么做。

清单 5.17　浏览器历史栈用 JavaScript 写的样子

```
// 以一个空数组开始，因为还没有浏览任何页面
var browserHistory = [];

// 浏览'https://www.google.com'
browserHistory.push('https://www.google.com');
/*
  ['https://www.google.com']
*/
```

```
// 搜索“droids”
browserHistory.push('https://www.google.com/#q=droids');
/*
  ['https://www.google.com',
  'https://www.google.com/#q=droids']
*/

// 点击一个搜索结果
browserHistory.push('https://en.wikipedia.org/wiki/Star_Wars:_Droids');
/*
  ['https://www.google.com',
  'https://www.google.com/#q=droids',
  'https://en.wikipedia.org/wiki/Star_Wars:_Droids'] */

// 挺有意思的，但不是你要找的机器人（droids），点击后退按钮
/*
  ['https://www.google.com',
  'https://www.google.com/#q=droids']
*/
```

树

树可能是计算机科学中最常见的数据结构，比你想象的更熟悉它。树实际上基于地上长的树命名，由主干、分支，以及叶子构成。你已经在 kittenbook 目录及 kittenbook.html 中见过树。kittenbook 目录是主干（树的所有部分都从主干长出来），其他目录是分支（分支可以包含叶子），而每个独立文件是叶子。在 kittenbook.html 中，<html>标签是主干，包含其他标签的标签是分支，剩余的标签就是叶子。标签中的标签叫作嵌套（nested，更像俄罗斯套娃，不像放鸟蛋的地方[1]）。

家谱最适合用树来表示，因为它们本来就是树形结构（见清单 5.18 和 5.19）。事实上，家谱中的一些名词也被用于计算机中的树。名词父母、孩子、祖先及兄弟都表现在树结构中。树结构非常合适表现层级关系。

清单 5.18 家谱以数组的数组形式表达

```
var familyTree = [ 'Abraham J. Simpson', 'Mona Simpson', [
   ['Homer Jay Simpson', 'Marjorie Jacqueline Simpson',
```

1 nest 在英文中除了表示嵌套，还有鸟窝的意思。

```
    ['Bartholomew Jo-Jo Simpson',
     'Lisa Marie Simpson',
     'Margaret Evelyn Simpson']
   ]
  ]
];
```

清单 5.19 家谱以对象形式表达

```
var familyTree = {
  father: 'Abraham J. Simpson',
  mother: 'Mona Simpson',
  children: [
    {
      child: 'Homer Jay Simpson',
      spouse: 'Marjorie Jacqueline Simpson',
      children: [
        {
          child: 'Bartholomew Jo-Jo Simpson'
        }, {
          child: 'Lisa Marie Simpson' },
        {
          child: 'Margaret Evelyn Simpson'
        }
      ]
    }
  ]
};
```

图

图是一组相连的元素，比如一组通过道路相连的城市，或者一群通过关系相连的朋友。图中的元素被称为节点，连接被称为边。图很有趣的一点是连接和节点一样重要。两个城市之间的道路和城市本身一样重要。图在社交网络中大量使用，定义人之间的连接、地点之间的连接、公司之间的连接等。互联网本身是一个通过链接将网页连接起来的巨大图。图 5.11 展示了一个简化了的内华达州道路图，清单 5.20 展示了那些相连的城市如何用一个图来表达。

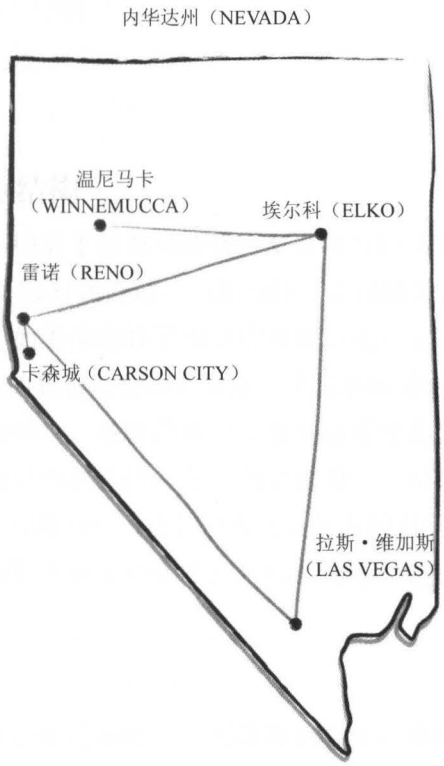

图 5.11 内华达州简化地图

清单 5.20 用图表达内华达州的城市

```
var nevadaCities = [
  {
    city: "Carson City",
    neighbors: [
      {
        city: "Reno",
        distance: 32
      }
    ]
  },
  {
    city: "Elko",
    neighbors: [
      {
```

```
      city: "Las Vegas",
      distance: 435
     },
     {
      city: "Reno",
      distance: 289
     },
     {
      city: "Winnemucca",
      distance: 124
     }
    ]
   },
   {
    city: "Las Vegas",
    neighbors: [
     {
      city: "Reno",
      distance: 452
     },
     {
      city: "Elko",
      distance: 435
     }
    ]
   },
   {
    city: "Reno",
    neighbors: [
     {
      city: "Carson City",
      distance: 32
     },
     {
      city: "Las Vegas",
      distance: 452
     },
     {
      city: "Elko",
      distance: 289
     }
    ]
   },
   {
    city: "Winnemucca",
    neighbors: [
     {
```

```
      city: "Elko",
      distance: 124
    }
  ]
}
];
```

如何选择高效的数据结构

你所选择的数据结构对于你要写的代码复杂度有巨大的影响。我们已经在尝试更新 author 的 age 标签时体会过，当你选择了错误的数据结构，最终你不得不在代码中费力解决它；但如果你选择了高效的数据结构，它会让你的代码更简洁、清晰。所以如何选择高效的数据结构呢？这是一个你会随着时间逐渐积累的一项技能，你会通过实践和失败来学习。重要的第一步是要知道你在选择数据结构，并且知道你可以从哪些数据结构中做出选择。在我创建的第一个网站，几乎所有的数据结构都用字符串来表示，因为我不知道更好的方式。某天我发现了 JSON，以及对象的力量；从那天开始，我的生活变得简单多了，我的代码也变得更简洁、易懂。

数据库

你程序中使用的数据都是临时的，存储在内存中，你的程序运行多久，数据就存在多久。当你需要数据保持更久时，数据库是一个非常好的选择。

长期（持久化）存储

有很多种办法可以永久存储数据。选择哪一种方法，依赖于你使用的编程环境，以及你需要存储的数据特性。对于少量的简单数据，你的程序可以在计算机的硬盘上创建一个新文件，然后把数据写到这个文件上。随着数据越来越多，越来越复杂，使用一种更健壮的解决方案，比如数据库，就变得很必要。当你的编程环境是浏览器中的一个网页，可选择的就没那么多了。非常少量的数据可以用 cookie 存储，但是大部分需要长期存储的数据必须被发送回服务器。

关系型数据库

关系型数据库是存储大量复杂数据最常用的方法之一。除了存储数据，关系型

数据库还可以组织数据，并创建一种结构化的方式来获取你需要的数据。关系型数据库并不是很灵活，所以在具体实现数据库之前，你需要花些时间来计划和设计它。为了更明确地说明这些抽象概念，我们仔细看一个练习，是一个给图书馆设计的数据库。

表

像我们之前用过的数据结构一样，数据库也是用于建模现实世界中的某些东西。我们这里会建模一个图书馆，所以我们要考虑应该在模型中包含数据库的什么实体。我们需要数据库了解并保存什么类型的数据？这里有一个图书馆数据库应该保存项目的不完全列表，但肯定会有更多：

- 书
- 作者
- 顾客
- 借出记录
- 雇员

每一类项目在数据库中都会表现为一张表。书的表对于图书馆中的每本书都有一行（成为记录）与之对应。表的列叫作字段，每个字段描述一个属性。书的表可能包含书的 ID、出版社、发行时间、体裁，以及页数等字段。

关系

关系型数据库的强大之处在于表之间存在着关系。关系型数据库有三类重要的关系（见图 5.12）。

- 1 对 1：表 A 中的一个元素与表 B 中的一个元素正好对应。1 对 1 关系比较少见，因为这两张表描述的是同一个东西，所以它们可能就是一张表。
- 1 对多：表 A 中的一个元素可以与表 B 中的一个或多个元素关联。顾客表与借出记录表存在一对多关系，因为一次借出记录属于且仅属于一个顾客，但一个顾客可以有很多个借出记录。
- 多对多：表 A 中的一个元素可能与表 B 中的一个或多个元素关联，而表 B 中的一个元素可能与表 A 中的一个或多个元素关联。作者表和书表之间就存在多对多关系，因为一本书可能有很多作者，而一个作者可能写过很多书。

不同表记录之间的关系通过一张表中的记录引用另一张表中记录的标识来建

立。例如，顾客表和借出记录表存在一对多关系，所以接触记录表会有一个叫作 PatronID 的字段，标识顾客表中拥有这条借出记录的顾客记录。借出记录表只需要 PatronID，指向顾客表中对应的记录，而不需要在借出记录表中重复记录顾客表中的所有信息。

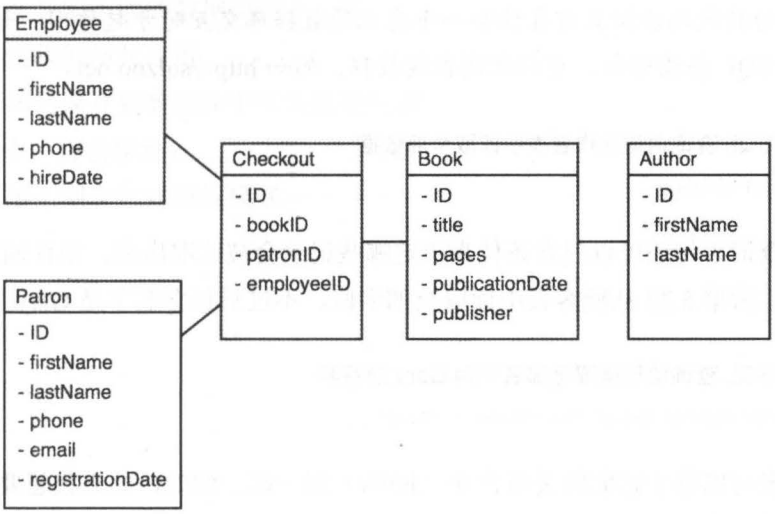

图 5.12 图书馆数据库架构

噢，CRUD！

对于完整的持久化存储系统（比如数据库），它们需要 4 个基本函数：创建、读取、更新和删除 CRUD。如果数据在系统中重复出现，要做 CRUD 几乎不可能。例如，想象一个顾客的电话号码记录在顾客表中，同时还记录在借出记录表中对应顾客的每一次借出记录中。现在如果一个顾客要更新数据库中的电话号码，数据库将不得不更新一次顾客表和很多次借出记录表。为了 CURD，请好好设计你的数据库，以便数据不会重复。

SQL 简介

关系型数据库有一种专用语言，叫作 SQL（结构化查询语言，Structured Query Language），用来创建、读取、更新和删除记录。SQL 的基础知识简单明了，而且我觉得很有意思。SQL 由结构化查询语句（正如其名）构成，而且这个结构有严格的

定义。要从数据库中读取数据，你的查询语句必须以单词 SELECT 开头，必须声明哪些字段是你想要的（如果你想要全部字段，就用*），并且必须包含你要读取的表名字。清单 5.21 展示了从作者表中读取全部数据。

> **注意**
>
> 下面的代码示例只有在你和一个真正的数据库交互时才起作用，如果你想要练习写 SQL 查询语句，可以试试在线教程，例如 http://sqlzoo.net。

清单 5.21　SQL 查询语句从作者表中读取全部数据

```
SELECT * FROM Author
```

一条查询语句还可以包含条件来决定哪些记录会被读取出来，条件通过 WHERE 语句表示。清单 5.22 从顾客表中读取全部字段，不过只包含名字是 Gary 的记录。

清单 5.22　SQL 查询语句读取全部名字为 Gary 的顾客

```
SELECT * FROM Patron WHERE firstName = 'Gary'
```

两张表可以基于它们的关系合并（JOIN）到一起。例如，如果我想要 ID 为 42 的借出记录的全部信息，我需要来自借出记录表和顾客表的数据（见清单 5.23）。关联表的数据可以使用 JOIN 语句结合到一起。

清单 5.23　SQL 查询语句读取借出记录为 42 的顾客的名和姓

```
SELECT firstName, lastName
       FROM Patron
       JOIN Checkout ON (Patron.id = Checkout.patronId)
       WHERE Checkout.id = '42'
```

SQL 可以做的事情比我们这里讨论的要多得多，而且精心设计一条高效的 SQL 语句来找到你需要的数据就像是解决一个有趣的谜题。如果你有兴趣学习更多 SQL 的知识，你可以找到很多在线交互式教程。我所学习的 SQL 知识大部分来自于 http://sqlzoo.net 上面的教程。

总结

本章介绍了很多概念，都是编程基础中的关键部分。你所选择的存储、组织及使用数据的方式，是软件高效运行的决定性因素。

本章你学到了：

- 是什么导致不同的数据类型不一样
- 数据结构如何起作用
- 数据如何保存到数据库中作为长期存储

下一章中，你会学到：

- 如何用正则表达式查找模式

6

正则表达式

> 注
> **项目：**使用正则表达式验证手机号码。

大部分编程入门书籍不会用一整章介绍正则表达式，因为正则表达式看起来比较吓人。结果，很多程序员永远不会学习正则表达式。尽管其他工具可以做很多正则表达式能做的事，但它们都不如正则表达式做得好。比如，你可以用螺丝刀的手柄钉钉子，但用锤子要更容易。对于某些工作，只有正则表达式才能胜任。尽早习惯使用正则表达式会对你的编程生涯带来好处。

话说到这，我需要提醒一句，正则表达式很难。它们非常简练，并且可能让人相当困惑。它们很难记录，而且很难读懂，也很容易出错。然而，它们相当有用。

Ctrl+F 组合键：寻找模式

上一章简要介绍了正则表达式，但你可能还想知道它们到底是什么以及怎么用。正则表达式很像 Word、谷歌浏览器，以及 Sublime Text 中的查找功能（键盘快捷键：Ctrl+F 组合键）。正则表达式也是在一组字符结合中搜索字符串，但它要强大得多。想象一下，你有 300 页文档，想要从中找到提到了年份（比如 2014）的所有地方。使用查找功能，你必须分别检索每一年。首先，你会搜索 1900，然后 1901，然后 1902 等，这会花费相当长的时间，直到你最终放弃。用正则表达式，你可以使用模式来搜索，一个年份就是一个非常简单的模式：一串由 4 个从 0 到 9 的数字组成的序列，见清单 6.1。

清单 6.1　使用正则表达式搜索年份

```
var yearPattern = /[0-9]{4}/
```

如你所见，正则表达式的语法有点奇怪，但如果仔细观察，你应该可以猜出大概。从 0 到 9 的数字表示为[0-9]，个数 4 的要求用{4}来表示。所以 yearPattern 就是在说，“任意从 0 到 9 的数字，重复 4 次”。完美的正则表达式会在字符串中匹配你想要找到的每一个东西，并且不匹配任何你不想要的东西。例如，yearPattern 会匹配任意 4 个数字序列，所以它会匹配所有的 4 位数年份（很好），但它还会匹配电话号码，比如 800-555-3456（不好），因为电话号码中也有一个 4 位数字。在本章中，你会学到需要的工具，将 yearPattern 变成更好的正则表达式。

在 JavaScript 中使用正则表达式

正则表达式要用字符串来测试，我们首先用一个简单的字符串和一个简单的正则表达式。在任意网页打开 Chrome 开发者工具，用你的名字创建一个字符串；然后用你的名字创建一个正则表达式。见清单 6.2。

清单 6.2　我的名字作为正则表达式

```
> var myName = 'Steven Foote';
  "Steven Foote"
> var namePattern = /Steven/;
  /Steven/
```

现在你需要检验这个字符串是否匹配正则表达式。你可以用几种不同的方式，每一种都有略微不同的目的和输出。

- test 返回布尔值，如果字符串匹配返回 true，否则返回 false。
- exec 要么返回一个数组，包含字符串中第一个匹配的部分，要么就什么都不返回。
- match 要么以数组形式返回字符串中匹配的部分，要么什么都不返回。match 和 exec 不同之处在于这个方法作用在字符串上，而正则表达式作用在参数上（见清单 6.3）。

清单 6.3 三种方式测试正则表达式/字符串组合（在控制台中）

```
> namePattern.test(myName);
  true
> namePattern.exec(myName);
  ["Steven"]
> myName.match(namePattern);
  ["Steven"]
```

重复

正则表达式 namePattern 一点都不像个模式；它跟使用查找功能没什么区别。如果我们想要匹配 Steven 或者 Steve 该如何做呢？清单 6.4 展示了 namePattern 的不灵活之处。

清单 6.4 尝试匹配 Steve

```
> var myNickname = 'Steve Foote';
  "Steve Foote"
> namePattern.test(myNickname);
  false
```

我的正则表达式太不灵活了，甚至连我的昵称都匹配不到。我们可以改进这一点——但丑话说在前面，这是正则表达式看着变得怪异的开始。

?

忘掉所有你关于问号含义的先入为主的知识，在正则表达式的世界里，一些日常生活中低调的字符可能会超越其本来扮演的角色，具有全新的、更有意义的能力。这个简单的?具有解决我们昵称问题的能力，见清单 6.5。

清单 6.5 强大的“?”

```
> namePattern = /Steven?/;
  /Steven?/
> namePattern.test(myNickname);
  true
> namePattern.text(myName);
  true
```

神奇！这个“?”告诉正则表达式，在“?”之前的字符应该至多出现一次，换

句话说，新的 namePattern 表示 n 可选。

+

现在你可能想“哇！正则表达式太棒了！”当然，你可能并没有那么激动，但是猜猜怎么着？不管你有多激动，正则表达式都能搞定。在正则表达式的王国里，“+”告诉正则表达式匹配一次或多次前导字符（就是“+”前面的那个字符）。见清单 6.6。

清单 6.6　正则表达式可以处理各种程度的激动

```
> var excitementPattern = /Wo+w/;
/Wo+w/
> var notYetExcited = 'Wow... Regular Expressions are weird :/';
"Wow... Regular Expressions are weird :/"
> excitementPattern.exec(notYetExcited);
["Wow"]
> var startingToGetExcited = 'Wooow. Regular Expressions are... pretty cool.'; "Wooow.
Regular Expressions are... hard."
> excitementPattern.exec(startingToGetExcited);
["Wooow"]
> var totallyLovingRegEx = 'Wooooooooooww! Regular Expressions are awesome!!!';
"Wooooooooooww! Regular Expressions are awesome!!!"
> excitementPattern.exec(totallyLovingRegEx);
["Wooooooooooww"]
> var excitedButBadAtSpelling = 'Www! Relugar Expresions, I <3 u!';
"Wwwwww! Relugar Expresions, I <3 u!"
> excitementPattern.exec(excitedButBadAtSpelling);
null
```

*

星号告诉正则表达式去匹配前导字符零次或多次。它绝对是正则表达式里面的摇滚明“星”。我们可以让 excitementPattern 表达式足够弹性，甚至可以处理拼写错误。见清单 6.7。

清单 6.7　正则表达式摇滚明“星”

```
> excitementPattern = /Wo*w/;
/Wo*w/
> var excitedButBadAtSpelling = 'Www! Relugar Expresions, I <3 u!';
"Wwwwww! Relugar Expresions, I <3 u!"
> excitementPattern.test(excitedButBadAtSpelling);
true
```

特殊字符和转义字符

现在你见识到了一些正则表达式王国的超级字符，你可能想要建立一个正则表达式来匹配包含这些神奇字符的字符串，来展示你是它们的大粉丝。见清单 6.8。

清单 6.8 查找星号及其他

```
> var starPattern = /*/;
SyntaxError: Unexpected token ILLEGAL

> var plusPattern = /+/;
SyntaxError: Invalid regular expression: /+/: Nothing to repeat
```

好吧，工作得不太顺利。第一个 SyntaxError 看起来有点神秘，不过第二个错误有些帮助。我们的 plusPattern 正则表达式不合法，因为“+”没有前导字符可以被重复。但是我们是想要匹配“+”，而不是重复某些字符。我们需要告诉正则表达式，我们想要匹配一个字符的前导值是“+”，而不是使用“+”的超能力。还记得我们在第 1 章“‘Hello，World!’写下第一个程序”中，字符串遇到撇号是怎么处理的吗？当时我们需要一个转义字符，现在也是。正则表达式也使用反斜杠表示转义。见清单 6.9。

清单 6.9 查找星号并真正找到它们

```
> var starPattern = /\*/;
/\*/
> var aStar = 'Kleene, you\'re a *';
"Kleene, you're a *."
> starPattern.test(aStar);
true
```

{1, 10}：创造属于你的超能力

你刚刚学到的那些特殊字符，是正则表达式中常用的，但还不够。如果“?”，“+”及“*”还不够特别，你可以用花括号来定义你需要的长度。还记得 yearPattern 用来匹配一个刚好 4 个字符的序列吗？你可以以 3 种方式使用花括号：

- {n}匹配刚好 n 个前导字符。
- {n,}匹配至少 n 个前导字符。
- {m,n}匹配至少 m 次，至多 n 次前导字符。

实际上，你可以用花括号语法达到“+”，“?”，以及“*”同样的能力。“?”等于{0, 1}，“+”等于{1,}，“*”等于{0,}。在本章中，我们会大量使用花括号。

匹配任意字符的“.”

有时你需要一个模式来匹配任意字符。例如，我们可以让 excitementPattern 只匹配包含某些形式的“Wow”及感叹号“!”的字符串，但在“Wow”和“!”之间可能有些我们并不关心的字符。正则表达式中另一个超能力特殊字符就是用来处理这种情况的：点号“.”，匹配任意字符。它就像正则表达式中的通配符。你可以组合点号和星号来匹配零个或多个字符，不管这些字符是什么。见清单 6.10。

清单 6.10　匹配任意字符

```
> excitementPattern = /Wo*w.*!/;
/Wo*w.*!/
> excitementPattern.test(notYetExcited);
false
> excitementPattern.test(excitedButBadAtSpelling);
true
```

随着我们的模式越来越复杂，很有必要可视化展示正则表达式代表的意思。www.regexper.com/是一个非常棒的在线工具，用来创建任意正则表达式的可视化图形。图 6.1 是 excitementPattern 的可视化图形，以“铁路”图的格式展示出来。火车始发于最左侧的点，每一个方块就像是火车站。每次火车停到某一站，字符串中的一个字符就会下车，并且字符必须按照正确的顺序下车。火车站的标签标示哪一类字符允许下车。如果火车停在驿站，而下一个字符不允许下车，火车就会脱轨，意味着字符串不匹配正则表达式定义的模式。

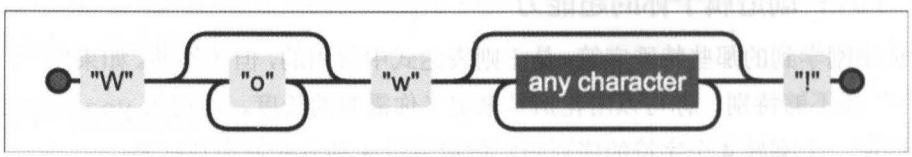

图 6.1　/Wo*w.*!/的图形化表示

不要太贪婪

正则表达式采取的是贪婪策略，不是电影《华尔街》中 Gordon Gekko 那种贪婪，

不过本质上还是一样。正则表达式的贪婪策略是指一个正则表达式会尽其所能地匹配字符串。比如给定一个字符串'Www! Regular Expressions, I <3 u!'，正则表达式excitementPattern(/Wo*w.*!/)会匹配整个字符串，而不仅仅是Www!的部分。这是因为.*!会让正则表达式匹配所有字符（包括感叹号），直到它找到最后一个感叹号。有时候贪婪是好的，但如果你不想要贪婪策略，在*或者+后面加一个+（喔?一定有很多超能力），它的贪婪策略就停止了。

从[A-Za-z]理解方括号

当你要匹配任意字符时，强大的“.”非常有用；但有时“任意字符”太宽泛了，使用方括号，可以让你明确定义你想要匹配哪些字符。如果我想要匹配我的名字，不管S是否大写，我可以用方括号来匹配S或者s，见清单6.11。

清单6.11 使用方括号处理大小写

```
> lowerCaseName = 'steven foote';
  "steven foote"
> namePattern // 记住名字模式的样子
  /Steven?/
> namePattern.test(lowerCaseName);
  false
> namePattern = /[Ss]teven?/;
  /[Ss]teven?/
> namePattern.test(lowerCaseName);
  true
```

字符列表

使用方括号最简单的方式是在方括号中列出所有接受的字符，在更新版的namePattern中，接受的字符是S和s，所以这两个字符都在方括号中。如果你想要匹配双引号或者单引号，你可以用[“‘]。如果你想要匹配任意小写的英文字母，你可以用[abcdefghijklmnopqrstuvwxyz]；而如果你想要匹配任意小写或大写英文字母，你可以用[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]。哇哦，太丑了，肯定有更好的方式。

范围

确实有更好的方式！使用 yearPattern，我们可以用[0-9]匹配任意从 0 到 9 的数字，这要比使用[0123456789]方便，尽管它们做的事情一样。方括号中，在两个字符之间的横线称为范围。范围最常用在表示数字范围和字母范围中。范围用在其他字符中会比较让人困惑，所以我不推荐这么用。你可以用[a-zA-Z]匹配任意大写或小写的英文字母。这要比之前的好太多了（而且你不需要担心错过一两个字母）。范围不一定要以 0 或者 A 开始，也不一定要以 9 或者 Z 结尾。比如，如果你想要只匹配字母表中的后一半字母，可以用[n-z]。你还可以组合使用独立字符和范围：[12357a-z]。

有时你可能想要在字符列表中包含一个横线，但横线在方括号中具有超能力，所以你需要使用……，你需要使用一个转义字符。像/[a-z]\-/这样的模式会匹配横线或任意小写字母。你还可以一起使用方括号和重复功能。yearPattern 使用方括号表示它要匹配数字 0-9；然后使用重复功能表示它要匹配那些数字 4 次。

排除

如果你要匹配除一些字符外的任意字符，正则表达式用另一个具有超能力的字符帮你做到。在一组字符前面加上一个脱字符（^）可以匹配除这组字符外的任意字符。假如说我坚持让别人叫我 Steve 而不是 Steven。事实上，我愿意在名字的最后使用任何字符，只要不是 n 就行（其实我更愿意被称为 Steven，但为了举这个例子，我们假设是这样）。新的 namePattern 就看起来像清单 6.12。

清单 6.12　不要叫我 Steven

```
> namePattern = /Steve[^n]/;
  /Steve[^n]/
> namePattern.test('Steve Foote');
  true
> namePattern.test('Steveq Foote');
  true
> namePattern.test('Steven Foote');
  false
```

电话号码模式

现在，你前面所学到的知识已经足够做一个验证电话号码的项目了。首先，我们需要考虑几种不同的合法电话号码格式。这里有几种可能的格式：

- 1-555-867-5309
- 555-867-5309
- (555)867-5309

我们可以在一个叫 regexpal 的在线正则表达式工具网站 http://regexpal.com/上实验这三种格式。在上面的输入框中，我们输入正则表达式（使用这个工具的时候不需要“/”），然后在下面的输入框中，我们输入想要去匹配的字符串，每行一个。把这三种格式分别输入到下面的框中。你还可以添加一些不能匹配这个模式的字符串，比如'Regular Expressions are great'和'-0-1-2-3-4-5-6'。这些显然不是电话号码，所以如果我们的模式能够匹配他们，我们就有事情做了。

在上面的每一种格式中，我们都看到一些数字和一些横线。在其中的一个，我们还看到括号。这些格式的长度在 12 到 14 个字符之间。我们可以用方括号和重复模式把这个变成基本正则表达式。清单 6.13 展示了这个正则表达式，图 6.2 是这个正则表达式的可视化展示。

清单 6.13 电话号码正则表达式，第一回合

```
var phoneNumberPattern = /[0-9\-\(\)]{12,14}/;
```

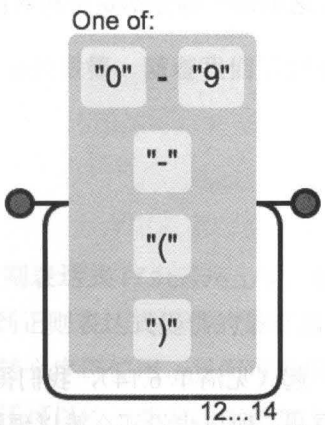

图 6.2 phoneNumberPattern 的可视化展示

我们的第一次尝试非常好，它匹配了所有三种格式，还匹配了一个不是电话号码的字符串（见图 6.3）。还不错，但我们可以做得更好。

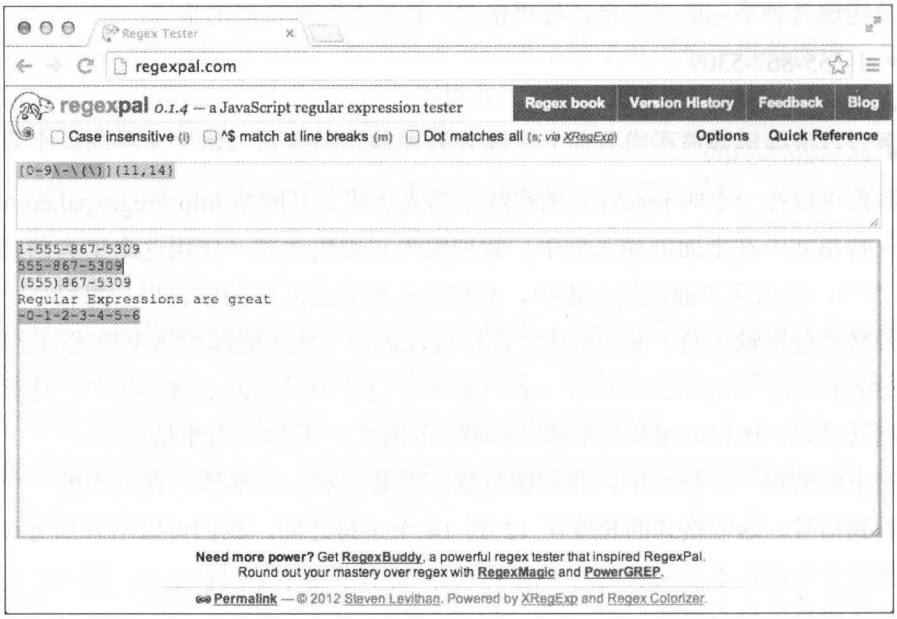

图 6.3　勇敢的第一次尝试，但我们还有提升空间

要让我们的模式更好，我们需要把电话号码分成几部分：1）一个可选的前缀：1-；2）一个可选的括弧；3）一个三个数字的区号；4）横线或者括回；5）三个数字；6）横线；7）4 个数字。这里每一部分都是个相对简单的模式：

1. `1?-?`
2. `\(?`
3. `[0-9]{3}`
4. `[\-\)]`
5. `[0-9]{3}`
6. `-`
7. `[0-9]{4}`

把这些简单的模式放在一起（见清单 6.14），我们得到一个不那么简单的模式，但我们知道每一部分代表的意思，所以也没那么难以理解。借助于图 6.4 的“铁道示

意图”，事情变得更加容易；尽管难以真正测试我们的新模式是否能匹配所有它应该匹配的内容，并且拒绝任何它不该匹配的内容，见图 6.5。

清单 6.14 电话号码正则表达式，第二回合

```
var phoneNumberPattern = /1?-?\(?[0-9]{3}[\-\)][0-9]{3}-[0-9]{4}/;
```

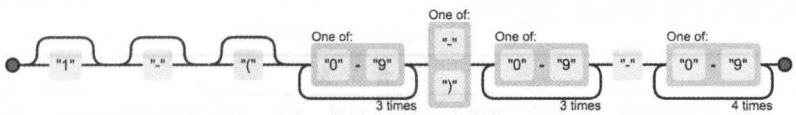

图 6.4 更好的电话号码正则表达式的可视化展示

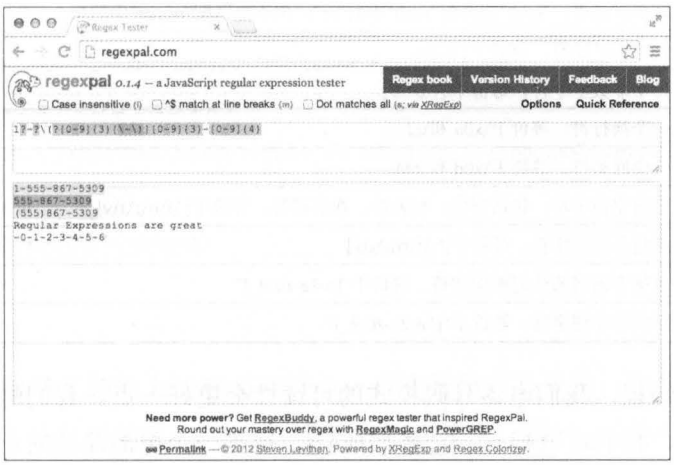

图 6.5 更好的电话号码正则表达式

这个电话号码表达式看起来很棒。随着我们学到更多正则表达式的功能，我们会让它变得更好。

我需要\s

相比我们写的第一个正则表达式(/Steven/)，我们已经对这个模式做了很多改进，而我们甚至还没有用到正则表达式提供的最有用的功能之一——反斜杠。到目前为止，我们把反斜杠当作转义字符使用，我们可以用它将特殊字符的超能力移除。但反斜杠还能做得更多，它还可以给一些普通字符加上超能力。

方括号的快捷方式

最常见的情况是，反斜杠可以将一个字母变成一组字符。举例来说，\s 匹配任意类型的空白字符，比如空格、制表符，和换行符（以及其他）。反斜杠还可以表示不能被直接用在正则表达式中的字符。例如，正则表达式通常全部写在一行，要表示一个换行符，你需要使用\n，因为你不能直接输入回车。下面的表格列出了左右可以通过反斜杠表达特殊意义的字符。

符　号	含　义
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”
\B	匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”
\d	匹配一个数字字符。等价于[0-9]
\D	匹配一个非数字字符。等价于[^0-9]
\n	匹配一个换行符。等价于\x0a 和\cJ
\r	匹配一个回车符。等价于\x0d 和\cM
\s	匹配任何空白字符，包括空格、制表符、换页符等。等价于[\f\n\r\t\v]
\t	匹配任何非空白字符。等价于[^ \f\n\r\t\v]
\w	匹配包括下画线的任何单词字符。等价于“[A-Za-z0-9_]”
\W	匹配任何非单词字符。等价于“[^A-Za-z0-9_]”

有了这些知识，我们电话号码模式的可读性会更好一点。我们的模式现在是重复三次[0-9]。我们可以把[0-9]替换成\d，得到一个更加简洁的正则表达式，而不改变其做的事情（见清单 6.15）。

清单 6.15　电话号码模式，第三回合

```
var phoneNumberPattern = /1?-?\(?\d{3}[\-\)]\d{3}-\d{4}/;
```

我们来添加一些代码到我们的 kittenbook 项目中。在 prompt.js 中，我们让用户提供电话号码而不是名字。然后我们用正则表达式来校验电话号码（见清单 6.16）。如你在第 2 章“软件如何工作”中学到的，我们可以发送一条短信到这个电话号码来进一步确认，然而，这部分超出了本书范围。现在我们先集中精力在使用正则表达式上做校验。

清单 6.16 promo.js 现在让用户输入并校验电话号码

```
// 获取用户名
var userName = prompt('Hello, what\'s your name?');
// 获取用户手机号
var phoneNumber = prompt('Hello ' + userName + ', what\'s your phone number?');
// 创建手机号模式
var phoneNumberPattern = /1?-?\(?\d{3}[\-\)]\d{3}-\d{4}/;
// 创建一个变量保存输出
var output = '<h1>Hello, ' + userName + '!</h1>';

// 电话号码是否有效?
if (phoneNumberPattern.test(phoneNumber)) {

  // 是的，电话号码有效！将成功信息加入到输出中
  output = output + '<p>' + kbValues.projectName + ' ' +   kbValues.versionNumber +
        ' viewed on: ' + kbValues.currentTime + '</p>';
} else {
  // 不是，电话号码无效。告诉用户问题是什么
  output = output + '<h2>That phone number is invalid: ' + phoneNumber;
}
// 将输出插入到网页中
document.body.innerHTML = output;
```

限制条件

在这些被赋予了超能力的普通字符中，有些是绝对必要的，因为它们代表了某个无法以其他方式在正则表达式中表达的字符（比如\n 和\t），然而，另外一些则只是用来让你少写代码的快捷方式（比如\w 和\s）。这些快捷方式很有用，前提是你明白它们的限制条件。\w 仅仅匹配英文字母，只要在你的字符串中加入一些西班牙语（如 olé ），正则表达式就会失败。另外，\s 还会匹配很多空白字符，你可能本来并不想要匹配它们。在这两种情况下，明确写出你想要匹配的字符（使用方括号）有时候是必要的。这些快捷方式很有用，除了不能用的时候。

(?:Groups)

我承认，到现在为止，正则表达式可能看起来有点无聊。尽管我不认为能够构造一个匹配任意格式电话号码的模式是件多么了不起的事，分组（特别是分组提取）让我真正爱上了正则表达式。分组就像大型正则表达式的子模式，而超能力符号可以处理整个分组。用几个例子来说明一下。假如你有一只叫作 fifi 的狗，你想要构造一个正则表达式来匹配任意含有狗狗名字的字符串。更复杂一点，我们假设你有时

会简称它为“fi”。你需要一个组，就是括在一对括号中的一些符号。一个非提取的组，也就是我们要用来寻找 fifi 的类型，以“?:”开头，见清单 6.17。

清单 6.17　寻找 fifi

```
> var fifiPattern = /(?:fi){1,2}/;
  /(?:fi){1,2}/
> var fifi = 'I have a dog named fifi';
  "I have a dog named fifi"
> var fi = 'Sometimes I call my dog fi';
  fi = "Sometimes I call my dog fi"
> fifiPattern.test(fifi);
  true
> fifiPattern.test(fi);
  true
```

如果我们要匹配'Stephen'，之前匹配'Steven'和'Steve'的 namePattern 会怎么样？我们可以用一个分组来解决，不过我们还要介绍另一个超级字符：“|”。这个“|”，或者叫作管道符（在退格键和回车键之间的反斜杠键上面），在正则表达式中，意味着“或”。管道符会拆分正则表达式，并匹配其左边或者右边的模式。例如，/apple|broccoli/ 可以匹配'apple juice'和'baked broccoli'，但不会匹配'carrot cake'。我们可以在一个分组中用一个管道符来匹配'Steven'中的 v 或者'Stephen'中的 ph，见清单 6.18。

清单 6.18　查找'Stephen'

```
> var namePattern = /Ste(?:v|ph)en?/;
  /Ste(?:v|ph)en?/
> var myName = 'Stephen';
  "Stephen"
> namePattern.test(myName);
  true
> myName = 'Steven';
  "Steven"
> namePattern.test(myName);
  true
```

最后一个例子，我们回想一下 phoneNumberPattern 中的前缀。大部分情况下都是对的，但有一个小问题。当我们把电话号码拆分成多个部分时，第一部分是一个可选的前缀“1-”，所以我们用了“1?-?”，问题是“1?-?”会匹配像“-877-555-1234”这样的电话号码，原因是这个模式会匹配零次或一次“1”，然

后匹配零次或一次“-”。我们其实是想要把它们放在一起匹配零次或一次，这可以用分组做到。见清单 6.19。

清单 6.19 修正前缀

```
var phoneNumberPattern = /(?:1-)?\(?\d{3}[\-\)]\d{3}-\d{4}/;
```

这样好多了。另一个要注意的地方是 phoneNumberPattern 中，在非分组的括号前，需要加上反斜杠。现在你知道了括号是特殊字符，那些反斜杠应该看起来比较合理吧。

(capture)

对我个人而言，提取分组(capturing group)是正则表达式中最有趣和有用的功能。它们让你可以从匹配的字符串中抽取数据。例如，我们可以对 phoneNumberPattern 使用提取功能，来获取字符串中的区号部分。提取分组只需要用括号（不需要前面提到的像“?:”这类特殊字符），你需要用 match 或 exec 来获得被提取的组，见清单 6.20。

清单 6.20 提取区号

```
var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
```

现在我们有了区号，就可以为用户创建更个性化的消息。假设我们的用户住在某个区号代表的位置（手机在某种意义上破坏了这个假设，不过我们先不管它），你可以在 values.js 中给 kbValues 对象添加一个 areaCodes 对象，见清单 6.21。

清单 6.21 给 kbValues 添加一个区号对象

```
var kbValues = {
  projectName: 'kittenbook',
  versionNumber: '0.0.1',
  areaCodes: {
    '408': 'Silicon Valley',
    '702': 'Las Vegas',
    '801': 'Northern Utah',
    '765': 'West Lafayette',
    '901': 'Memphis',
    '507': 'Rochester, MN'
  }
};
```

你想在 areaCodes 对象中添加多少区号都可以。现在你可以更新 prompt.js，使用用户电话号码中的区号来创建一个更加个性化的问候语。第一步是更新正则表达式，加入对区号进行分组提取的代码（见清单 6.20）。然后你需要用 exec 或者 match 来提取匹配的部分，这会返回一个数组，区号就在索引为 1 的位置。见清单 6.22。

清单 6.22　提取区号

```
// 创建电话号码模式
var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
// 从 phoneNumber 获取匹配结果
var phoneMatches = phoneNumberPattern.exec(phoneNumber);
// 如果电话号码是 901-555-5309, phoneMatches 将是['901-555-5309', '901']
var areaCode = phoneMatches[1];
```

最后，你需要添加一些个性化消息到输出中。这会给我们带来一些有意思的问题。你知道如何获取一个对象的不同属性，使用对象名，然后一个点，接着是属性名，就像 kbValues.projectName。那么我们可以用 kbValues.areaCodes 获取区号对象，而我们需要的区号存储在 areaCode 变量中，但我们如何获取区号对应的值呢？我们可以试试 kbValues.areaCodes.areaCode，但这会在 areaCodes 中查找一个名为 areaCode 的属性。在 JavaScript 中，你可以使用方括号语法来读取一个对象的属性（见清单 6.23），看起来就像读取一个数组中的元素。清单 6.24 展示了如何使用方括号语法来基于区号读取位置信息。

清单 6.23　用方括号语法读取对象的属性

```
> var states = {'AL': 'Alabama',
                'AK': 'Alaska',
                'AZ': 'Arizona',
                'AK': 'Arkansas'};
> states.AL;
  'Alabama'
> var stateName = 'AL';
  'AL'
> states.stateName; // 这样不能工作
  undefined
> states[stateName];
  'Alabama'
> states['AL'];
  'Alabama'
```

清单 6.24 基于 areaCode 读取位置

```
// 使用方括号语法获取位置
var userLocation = kbValues.areaCodes[areaCode];
```

现在你可以用任何你喜欢的方式把 location 添加到输出中。也许你可以问用户 location 所在地最近的天气如何。无论如何，你的用户一定会因为你通过他们的电话号码查到他们住在哪儿而大吃一惊。如果他们知道你为了查到他们的位置而使用的各种正则表达式魔法，会更加吃惊。

提取标签

现在你已经见识了一些使用正则表达式和提取分组的方法，是时候让你自己去尝试一下了。在任意页面打开谷歌浏览器的开发者工具（建议打开 Mozilla Developer Network 这个网站，但任意页面都可以）。在“元素”标签页中找到一个 HTML 标签，复制元素的 HTML（见图 6.6），然后在控制台将字符串保存到一个变量中（建议用一个没有子元素的元素，这样你就不需要在保存字符串时处理换行符）。现在，写一个正则表达式，提取这个标签名（例如，<h2>的标签名是 h2）。你应该可以用同样的正则表达式来提取任意复制的 HTML 标签的标签名。

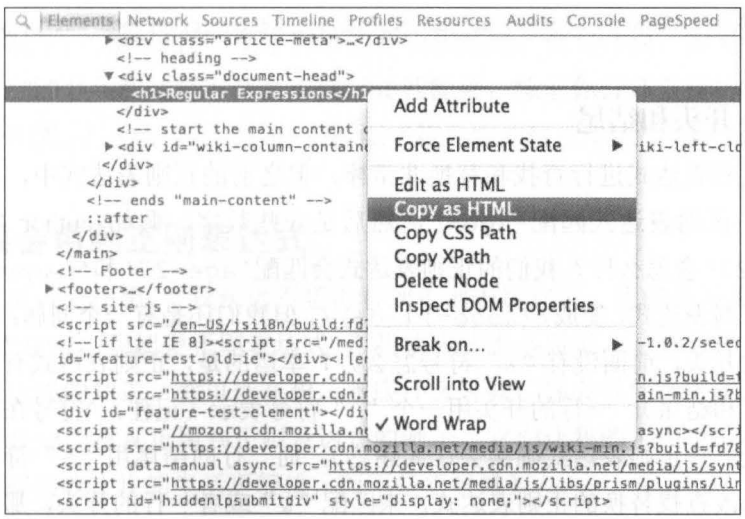

图 6.6 复制给定标签的 HTML

高级查找和替换

要看看提取分组功能其他有用的地方，我们可以回想下上一章的清单 5.9。我们在那用又长又复杂的代码来更新 author 字符串的年龄属性。我们可以用正则表达式来大幅度改进那段又长又复杂的代码，通过分组提取、高级搜索和替换就可以做到，见清单 6.25。

清单 6.25　用分组提取来改进字符串数据结构

```
var author = 'firstName=Steven&lastName=Foote&age=27&favoriteFoods=waffles,Thai
curry';
// 使用字符串内置的替换方法
author.replace(/(age=)(\d+)/, function(fullMatch, group1, group2) {
  /**
    * fullMatch 包含" age=27"  * group1 包含" age="
    * group2 包含" 27"
    */

  // 我们返回的内容会替换原始字符串的整个匹配
  // 对 group2 加 1，然后将两个组一起返回
  // parseInt 会将字符串 "27" 转换成数字 27，这样我们可以将它们相加
  // 如果没有 parseInt，"27" +1 会变成 "271"，而不是 28
Return group1 + (parseInt (group2, 10) + 1);
});
// Author 现在是'firstName=Steven&lastName=Foote&age=28&favoriteFoods=waffles,Thai curry'
```

（一行的）开头和结尾

使用正则表达式进行查找和替换非常棒，但之前的正则表达式中，有一部分不那么正确。正则表达式匹配'age='，然后是一些数字，假如 author 字符串还包括'page=23'会怎么样？我们的正则表达式会匹配'age=27'和'page=23'。我们可以修改正则表达式，变成/(&age=)(\d+)/，但我们还是有一个问题：假如'age'是字符串的开头，前面没有“&”符号怎么办？幸运的是，正则表达式有办法定位一行的开头（和结尾）。一行的开头用一个“^”符号表示（记住，^符号在方括号中有不同的意义——正则表达式确实会让人迷惑），而一行的结尾用“$”符号来表示。我们可以修改查找替换的正则表达式，来匹配“&”或者一行的开头，那么'age'就成了：/((?:^|&)age=)(\d+)/。

标记

正则表达式有一些默认行为，你可以通过标记来打开或者关闭。这些默认行为通常工作得很好，当他们不工作时，我们可以关掉它们。

全局匹配

默认情况下，一旦正则表达式在字符串中找到一个匹配的项，就会停止查找。全局标签可以告诉正则表达式去查找字符串中的全部匹配项，而不仅仅是第一个。要设置全局标签，你需要在正则表达式结束的斜杠后面添加一个“g”：`/[0-9]{4}/g` 会在字符串中查找所有的年份，而不仅仅是第一个。

忽略大小写

正则表达式默认大小写敏感，但有时你不关心文本是大写还是小写。你可以通过使用忽略大小写标签让正则表达式不要大小写敏感（用“i”来设置，所以 `/ste(?:ph|v)en/gi` 会找到字符串中的每一个'Steven','steven','Stephen'，和'stephen'）。注意全局匹配标签和忽略大小写标签可以（但不是必须）一起使用。

多行

默认情况下，^和$仅仅匹配字符串的开头和结尾。换句话说，如果你的字符串包含多行，行的开头和结尾不会真正被^和$匹配到，除非你打开多行标签。多行标签使用 m 来配置。`/.*\.html$/gm` 会匹配字符串中的每一个以'.html'结尾的行。

什么时候会用到正则表达式

现在你很可能在想，“我永远不想再看到正则表达式了，我的头好疼。”或者也许你在想，“哇哦，我超爱正则表达式！”我猜应该不是后者。不管怎样，你学到了很多正则表达式的知识，你可能在想，到底什么时候，你会以什么方式用到它们。下面的例子是一些我真正用到正则表达式的例子，它们让我的工作更轻松（真的，正则表达式确实能让我的生活更轻松，而不是更艰难）。

grep

第 3 章“认识你的计算机”中，你学到了命令行工具 grep。目前，我把 grep

当成同时搜索多个文件的查找工具。但是 grep 其实是基于正则表达式的（事实上，grep 就是全局正则表达式打印（global regular expression print）的缩写）。使用 grep，你可以同时在多个文件中按照模式进行搜索。当工作在一个有很多文件的项目中（LinkedIn 有非常非常多的文件）时，grep 是非常有用的工具。比如，我可以用一个 grep 命令，搜索全部 CSS 文件，查找使用“fload: left;”或者“float: right;”的所有地方。

代码重构

我在使用 HTML 写这本书，我用正则表达式（特别是高级查找和替换）来让我工作得更快。在我开始一个新的章节时，我会把关于这一章的提纲复制到一个新的 HTML 文件中，提纲中的项使用 HTML 列表（<ul>和<li>标签），但这些项在真正的章节中会变成标题（<h1>，<h2>等）。我可以手工修改每一个标签，但我选择写两个正则表达式来做这件事。见图 6.7，Sublime Text、Vim 及其他很多编辑器都在查找替换功能中支持正则表达式（以及分组提取）。

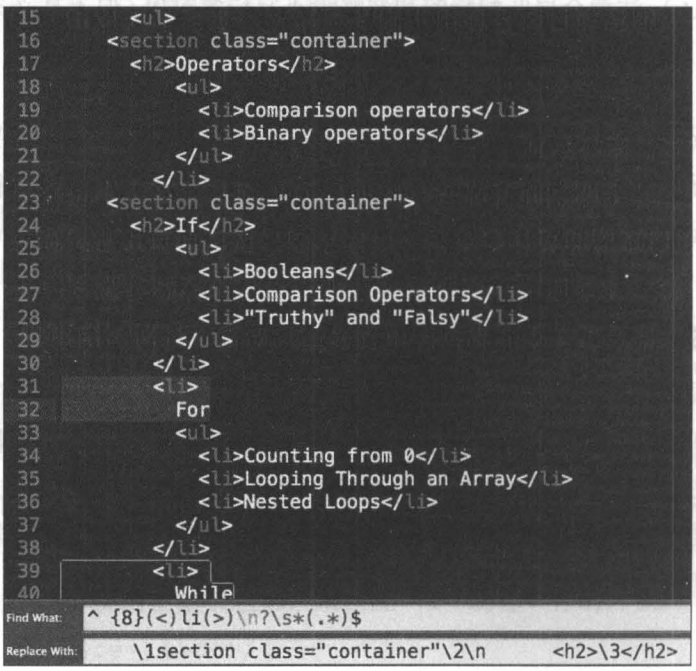

图 6.7　用正则表达式将列表项改成标题

校验

在本章中，你写过一个非常好的电话号码校验器（针对美国电话号码）。几乎每次我想要做检验，都会使用正则表达式。邮箱特别难校验；搜索一下“邮箱正则表达式”，你会明白我说的是什么意思。再次声明，仅仅通过电话号码或者邮箱地址符合正则表达式，并不能判断其就是合法的，但正则表达式至少可以告诉你这个字符串是否“看起来”比较合理。

数据抽取

我的第一个编程项目就是使用正则表达式从一个很大（1MB）的文本文件中抽取书的信息。那个项目可能是我特别喜欢分组提取功能的原因。当我看到正则表达式将一个完全不可用的30000页报告变成一个表格时，我就折服了。

总结

本章你学到了很多东西。曾经像是随机字符的东西（咳咳，`/(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/`，咳咳）现在对你来说应该看起来合理了。当然，你还是需要花些时间才能搞清楚它的意思，但你现在有能力搞清楚它。正则表达式是一个很棒的工具，但记住不能仅仅因为你拿了一个锤子，就把所有东西看成钉子。本章你学到了：

- 正则表达式语法
- 量词，比如`?`,`+`和`*`
- 方括号中的字符集
- 组和分组提取
- 高级查找替换，以及正则表达式的其他应用

下一章中，你会学到：

- 操作符
- 使用流程控制进行编程
- 如何使用`if`、`for`、`while`和`switch`来控制流程
- 使用`try`进行防御式编程
- 触发器和事件

7

何时使用 if、for、while

> 注
> **项目：** 将 Facebook 新鲜事中的每一个图片 URL 打印到控制台。

这是你期待已久的章节。到目前为止，你基本上都在学如何编程，但并没有学很多写代码的知识。你已经学过了创建一个真正可运行程序的技巧，但你还没写过多少代码。这一章全是关于如何写代码的。特别是，你会学习如何控制程序执行的流程。流程控制工具让你的程序决定该执行哪条指令，指令该执行多长时间，以及指令何时被执行。学习写代码是学习编程的基础部分，而流程控制是学习写代码的关键部分。所以，请集中精力！

操作符

多亏了小学数学，你可能早就非常熟悉操作符。加减乘除都是操作符，一个操作符会对其操作数执行特定的操作。在算式 20+22 中，20 和 22 是操作数，+是操作符，两数相加是执行的操作。你已经在我们写过的代码中见过一些操作符了。操作符分为几类。

比较操作符

比较操作符会比较两个操作数的值，返回一个布尔值（真——`true`，或者假——`false`）。有 6 个比较操作符，你可能除了其中一个，其他都在小学学过。

1. “相等”检查两个操作数是否一样。大部分语言中，这个操作符是“==”，在 JavaScript 中，最好使用“===”，原因会在后面解释。所以`'abc' === 123` 返回 `false`，`42 === 42` 返回 `true`。

2.“不相等”检查两个操作数是否不一样。不论相等操作符对于两个操作数返回什么值，不相等就会返回相反的值。在大部分语言中，不相等操作符是“!=”，但JavaScript不一样，它使用“!==”。所以'abc' !== 123会返回true，而42 !== 42会返回false。不相等操作符在我的小学课程没有教过。

3.“大于”检查操作符左边的操作数是否大于右边的操作数，而且操作数必须是数字。大于操作符使用熟悉的“>”符号（甚至JavaScript也用这个）。100 > 10返回true而-1 > 1返回false。

4.“大于等于”和“大于”操作符的效果一样，不过如果操作数相等它也会返回true：42 >= 42返回true。

5.“小于”和“大于”操作符相反：-1 < 1返回true。

6.“小于等于”应该不需要解释了：-1 <= 1返回true而42 <= 42返回true。

逻辑操作符

随着你开始使用三个逻辑操作符（与——and，或——or，非——not），你会改变思考和说话的方式，至少，我改变了。对我来说，“与”这个字不再是没有意义的连词，因为“与”有了特定的意义。例如，我们假设你站在雨中等公交车，假设你是站在一个小店门口。店主走到门口，对你说，“如果你感到冷与湿，欢迎你到里面来等。”（警告：你快要开始理解为什么程序员总是一副粗鲁不爱社交的样子了。）店主可能想表达的意思是说“如果你感觉冷或湿，欢迎进来等。”当然，店主（我们叫她苏）可能让你进去等，不管你感到哪一种情况。当然，她不会在门口把你拦下跟你说，“等一下，你只是湿了，看起来一点都不冷，在外面淋雨吧。”但一个程序员习惯了“与”的特定意义，所以程序员可能会倾向于按照字面意思理解苏，或者甚至纠正她。幸运的是，在我的经历中，极少数程序员真的会如此纠结于字面意思。

“与”操作符（JavaScript中是&&）有两个操作数，其检测这两个操作数是否都是真（true）。如果任何一个是假（false），“与”操作符返回假。“与”操作符会先计算其左侧的值，只有当左侧的操作数是真，才检测右侧。如果左侧操作数为假，“与”操作符已经知道这个语句的结果一定为假，不论右侧的操作数是多少，所以没有必要再去计算右侧操作数的值。见清单7.1。

图 7.1 刻板的程序员在努力不要像计算机一样思考（漫画，Dennis Hengeveld 绘，happysimpleton.com）

清单 7.1 “与”操作符

```
> true && true;
  true
> true && false;
  false
> false && true;
  false
> ('a' === 'b') && (3 > 2); // (3 > 2)永远不会被求值
  false
```

“或”操作符（JavaScript 中的||）也有两个操作数，任何一个为真，则返回真。和“与”操作符类似，“或”操作符会先计算左侧操作数的值。右侧的操作数只有当左侧的操作数为假时才会被计算。如果左侧的操作数为真，“或”操作数已经确定整个语句为真。见清单 7.2。

清单 7.2 “或”操作符

```
> true || true;
  true
> true || false;
  true
> false || false;
  false
```

```
> ('a' === 'a') || (3 === 2); // (3==2)永远不会被求值
  true
```

“非”操作符（JavaScript 的!）接受一个操作数，返回取反后的值。如果操作数为真，“非”操作符返回假，反之亦然。见清单 7.3。

清单 7.3　“非”操作符

```
> !true;
  false
> !false;
  true
> !('a' === 'b') // 这样会带来困扰。'a' !== 'b'做的是同一件事，看起来更合理
  true
```

一元操作符

一元操作符是指只对一个操作数起作用的操作符。你已经见过的“非”操作符就是一元操作符。另一个一元操作符的例子是自增操作符。自增操作符的操作数是一个数字，自增操作符会对这个操作数加一（见清单 7.4）。C++的名字就来自于自增操作符（C++和 C 很像，稍好一点）。

清单 7.4　使用自增操作符

```
> num = 2;
  2
> num++; // 和num = num + 1一样
  3
> num;
  3
> num--; // 还可以做减法
  2
```

二元操作符

目前为止，最常用的操作符还是二元操作符。二元操作符并不是只对二进制的 0 和 1 起作用的操作符，而是含有两个操作数的操作符。所有的比较操作符都是二元操作符。

强制转换

一些语言（包括 JavaScript）有时会在两个操作数类型不同时执行强制转换。这意味着其中一个操作数的数据类型会被改变，以执行操作符的操作。在尝试'2'+2时，你已经见识到 JavaScript 的强制转换。数字 2 会被强制转换成一个字符串，所以输出是'22'。类型强制转换在你知道自己在做什么时可能非常有用，但也可能在一些场景非常危险。如果你在 Perl 中尝试'2'+2，你会得到 4，而不是'22'，因为 Perl 会把字符串转换成数字，而不是数字转字符串。如果你在 JavaScript 中尝试'2' * 4，你会得到一个错误，而不是'2222'。

JavaScript 一个臭名昭著的特性就是在判断相等时使用类型转换。如果你用==（而不是===）来检测相等，JavaScript 会转换类型，清单 7.5 显示了==有多么奇怪。

清单 7.5 在比较操作时类型转换会让人迷惑

```
> '' == '0' // 没必要强制转换，等价于''==='0'
  false
> '' == 0   // 将''强转成数字会返回 0，
            // 所以这等价于 0 === 0
  true
> 0 == '0'  // '0'被强转成数字 0，
            // 所以这等价于 0 ===0
  true
```

操作顺序

和数学计算一样，操作执行的顺序在编程中是被明确定义的，这一点很重要。好消息是，数学操作的顺序和在进行数学计算时的顺序一样（乘法在加法执行前执行），但在编程中，还有一些别的操作符要学习。就像数学中，你可以用括号来强制让某些操作符和操作数结合在一起，说实话，在这一点上，我自己没怎么遇到过问题，但当我弄不清楚时，括号总是能解决问题。

赋值

将一个值赋给一个变量其实也是一个操作符——比较特殊的一个。大部分二元操作会先对操作符左边进行求值，然后对右边进行求值，但赋值操作符先从右边开始。如果你在语句中有个赋值，会先从右边开始，最后到左边。见清单 7.6。

清单 7.6 多个赋值

```
var x = 1;
var y = 2;
var z = x = y = x + y;
// x, y, z 都是 3 (1+2)
```

数学操作符

基本的数学操作都是二元操作。加法操作符是+，减法是-，乘法是*，除法是/。有些语言会包含一个指数运算操作符（Python 和 Perl 是**，Visual Basic 是^），但 JavaScript 并没有。除了指数运算，很多数学运算都不会使用特殊符号来表示（比如开方，对数和正弦函数）。大部分语言内建支持这些操作，如果你要用，只需要在搜索引擎中搜索一下“JavaScript 开方”就会有很多结果。

模运算

在我刚学编程的时候，除了取模操作外，所有的数学操作符都是我小学数学课上很熟悉的。“取模”这个名字看起来很奇怪，而且其使用的符号也很奇怪，但其实并不是一个奇怪的概念。取模操作会返回给你左侧操作数被右侧操作数除之后的余数。也就是说 2%2（读作“2 模 2”）是 0，而 3%2 是 1。取模操作在处理具有循环性质的数字时非常有用，比如时间。举个例子，假设今天是星期二，你的程序想要创建一个 85 天以后的事件，从今天开始的 85 天后是星期几呢？周四是一星期的第五天，一星期共有 7 天，所以 (5+85)%7 得到 6，也就是周五。取模运算在确定一个数字是奇数还是偶数时也很有用（见清单 7.7）。

清单 7.7　奇数还是偶数

```
// 请求一个数字
var favoriteNumber = prompt('What\'s your favorite number?');

// 将字符串转换成数字
favoriteNumber = parseInt(favoriteNumber, 10);

// 使用取模操作符判断数字是奇数还是偶数
if (favoriteNumber % 2 === 0) {
  alert('Your favorite number is even'); }
else {
  alert('That\'s an odd number, literally');
}
```

三元操作符

既然我们在说操作符，我们还是应该说说三元操作符。正如你猜测的，三元操作符接受三个操作数。不是所有语言都有三元操作符，因为三元操作其实就是简单的 if/else 语句（见 else if 章节）的省略写法。三元操作的语法是“操作数 A ? 操作数 B : 操作数 C”。如果操作数 A 为真，则计算操作数 B 的值；否则，计算操作数 C 的值。我们来看看清单 7.8 中的实际代码。

清单 7.8 趣味三段论

```
// 请求一个数字
var favoriteNumber = prompt('What\'s your favorite number?');

// 将字符串转换成数字
favoriteNumber = parseInt(favoriteNumber, 10);

var oddOrEven = ( (favoriteNumber % 2 === 0) ? 'even' : 'odd' );
alert('You\'re favorite number is ' + oddOrEven);
```

当你理解它在做什么时，不得不承认，三元操作符很方便，但是，它们也可能让代码很难懂，所以不应该被过度使用。

if

我们关于操作符长度的问题把我们带到这。if 语句是代码中流程控制的关键，程序的流程通常会用类似图 7.2 中的流程图来表示。每一个菱形包含了一个是非问题，也被称为条件，通过它们来决定该执行什么代码。清单 7.9 展示了一个简单的 if 语句用法，如果问题的答案是真，则 if 代码块中的代码会被执行。如果你想，你可以添加一个 else 语句，像清单 7.7 那样。只有当问题的答案是假，else 代码块中的代码才会被执行。

清单 7.9 简单的 if 语句

```
if (yesOrNoQuestion) {
  // if 代码块

  // 如果 yesOrNoQuestion 的答案是"yes"
  // 或者"true"，这里的指令就会被执行
}
```

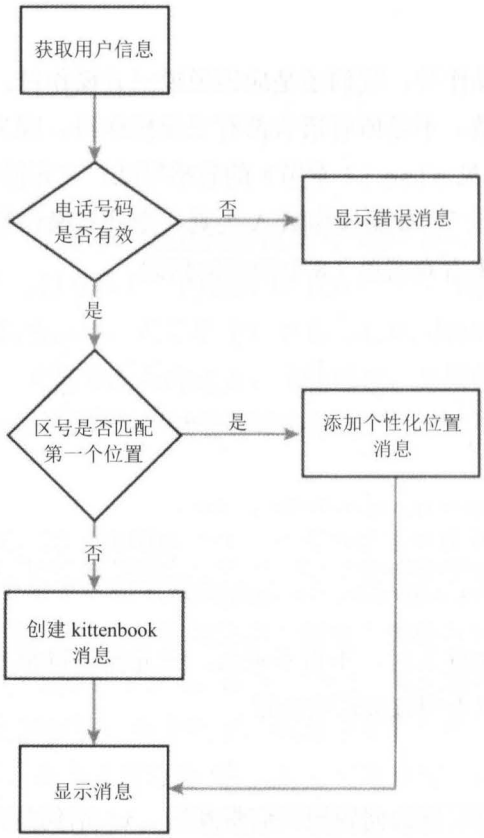

图 7.2　表示 **prompt.js** 中代码的流程图

else if

如果对于第一个问题的答案是“否”，可能需要问第二个问题，使用 `else if`，你就可以问第二个（以及第三个，第四个等）问题。对于流程控制来说，`else if` 非常重要。清单 7.10 是一个例子。

清单 7.10　基于身高你应该做什么运动，使用 `else if`

```
if (heightInFeet > 7) {
  // 你很高，所以篮球可能对你来说是个不错的职业
  learnToPlayBasketball();
} else if (heightInFeet < 5) {
  // 你可能成为一个伟大的赛马骑士
  learnToRaceHorses();
} else {
```

```
  // 你没有特别高或者特别矮，试试棒球
  learnToPlayBaseball();
}
```

“真”和“假”

if 语句接受是非问题作为参数的想法只是对了一半。如果给一个不是是非问题的东西，像清单 7.11，会怎么样呢？我们在控制台试试看。

清单 7.11 当 if 遇上字符串

```
> if ('true') {
    console.log('"true" is true');
  }
  "true" is true

> if ('false') {
    console.log('"false" is true(?)');
  }
  "false" is true(?)
```

'false'怎么会变成真呢？很多编程语言都有一个“真假性”的概念，意味着任何数据类型的值都能够用“真”或者“假”来表示。一个值的真假性由它在强制转换成布尔类型时的返回值所决定，要么是 true，要么是 false。每一个编程语言都自己定义了一套什么是“真”，什么是“假”的规则。通常来说，0 是“假”，而其他数字都是“真”；''（空字符串）是“假”，而其他字符串是“真”。因此，字符串'false'是“真”，并被强制转换成 true。真假性可以让你用布尔类型以外的数据类型作为 if 的条件。我们可以在 prompt.js 中使用真假性来检验用户是否输入了名字（见清单 7.12）。

清单 7.12 使用真假性改进 prompt.js

```
// 获取用户名字
var userName = prompt('Hello, what\'s your name?');

// 如果没有输入名字，再请求一次
if (!userName) {
  userName = prompt('You didn\'t enter a name. Really, what\'s your name?');
}
```

switch

在我们讨论 kittenbook 时，假设你想要 kittenbook 专门给名叫 Tyler 的人一个特殊的欢迎消息，你可以使用刚刚学到的 `if` 语句添加个性化消息。现在，假定你还想要添加一个特殊消息给名为 Jeremy，Kimberly，Lisa 和 Jason 的人，你可以用一堆 `if-else` 语句，但你的代码就会变得很长，而且在每一个 `else if` 中基本上都要判断相同的问题。不过，你可以使用 `switch` 语句，一个 `switch` 语句接受一个值，判断该值与一组预设的值是否相等。见清单 7.13。

清单 7.13 使用 `switch` 的特殊欢迎消息

```
var userName = prompt('Hi, what\'s your name?');
switch (userName) {
  case 'Tyler':
    greeting = 'Howdy!';
    break;
  case 'Jeremy':
    greeting = 'Cheers!';
    break;
  case 'Kimberly':
    greeting = 'Maayong aga';
    break;
  case 'Lisa':
    greeting = 'Bonjour';
    break;
  case 'Jason':
    greeting = 'Ola';
    break;
default:
    greeting = 'Hello';
    break;
}
```

`switch` 按顺序对比输入值（例子中的 `userName`）和每一个条件值，从上到下。如果该值匹配，则这个条件值后面的代码就会被执行。如果所有条件都不匹配，`default` 下面的代码就会被执行。注意在每一个 `case` 的最后都有一个 `break` 语句，如果你不加上 `break` 语句，下一个 `case` 中的代码就会被执行，即使这个值并不匹配那个条件。某种意义上，这是一个特性，因为你可以让多个条件执行一样的代码，然而，这个“特性”在你不注意的情况下会产生很多 bug。

“语法糖”

switch 语句和一堆 if-else 语句做的事情完全一样，不过更简洁。在编程中，你总是可以用很多不同的方式做同样的事，当一门编程语言给了以一种更简洁容易的方式来完成某个特定任务时，通常被称为语法糖（可能简洁的语法“味道”更好？）。几乎总是有另一种方式把你的代码以其他方式写出来，其他方式可能执行的更快，可能更容易理解，可能更安全，或者可能仅仅是“另一种”方式。不论如何，你的代码应该不是唯一的一种，所以，保持一个开放态度。

for

另一个控制程序流程的方式是使用循环。循环会一遍又一遍执行一组指令，直到你让它停下来。最常见的循环类型之一是 for 循环，其允许你定义循环指令具体执行多少次。for 循环的语法由四个部分组成：赋值、条件、增量和代码主体。for 循环还有一个迭代器，主要用来记录循环被执行了多少次。见清单 7.14。

清单 7.14 JavaScript 中的 for 循环

```
for (/* 赋值 */ /* 条件 */ /* 递增 */) {

  /*主体代码*/

}
```

在赋值部分，迭代器得到初始值。赋值只执行一次，即第一次 for 循环的迭代之前。迭代器是一个变量，变量名通常叫做 i。i 代表迭代器（iterator）？还是索引（index）？还是因为编程来源于数学？互联网上对此并没有明确的答案，但使用 i 作为迭代器已经是一个固定的范式，你应该遵守。

条件是一个是非问题，就像 if 语句中的条件，尽管 for 循环中的条件通常是大于或者小于。（一会儿你就会看到原因。）条件在每次迭代之前都会被执行，如果条件为假，循环结束。

增量用来改变迭代器的值。增量操作在每次迭代之后执行。增量操作通常会对迭代器加一，不过你想让它增加或者减少多少都可以。

代码主体包含一些在每个循环迭代中都要执行的指令。任何合法的指令都允许放在代码主体中，包括赋值语句，if 语句，甚至 for 循环。

清单 7.15 中的循环执行了 11 次。迭代器 i 以 0 开始，条件部分发现 0 小于等

于 10，函数主体执行，然后增量操作执行，迭代器被加一。循环一直重复，直到 i 的值变成 11，11 <= 10 返回 false，所以 console.log 没有被执行。还记得数组索引从 0 开始吗？可以看出，计算机科学中的数都从 0 开始，for 循环也不例外。

清单 7.15　for 循环例子

```
for ( var i = 0; i <= 10; i++ ) {
  console.log('The loop has executed ' + (i + 1) + ' times.');
}
```

循环遍历一个数组

for 循环在数组中遍历每一个值时特别有用。假设我们要打印所有 author 对象喜欢的食物，for 循环是最佳选择（见清单 7.16）。

清单 7.16　循环遍历一个数组

```
for ( var i = 0; i < author.favoriteFoods.length; i++ ) {
  console.log(author.favoriteFoods[i]);
}
```

在遍历数组时，迭代器 i 从 0 开始（数组第一个元素的索引），不断增加 1，直到迭代器比数组长度小 1。为什么在少了 1 的位置停下？因为在计算元素数量（数组长度）时是从 1（而不是 0）开始的。晕了吗？如果我们用<=而不是<来表示条件，我们会尝试读取一个数组中不存在的元素。在一些语言中，尝试读取超出数组边界的元素会得到一个错误。这个失误如此常见（足够让人困惑）以至于它有一个自己的名字：差一错误。

遍历图片

第 5 章“数据（类型）、数据（结构）、数据（库）”中，在 getImages.js 中，你创建了一个由 Facebook 图片组成的数组。本章你的挑战是遍历 images 数组的每一个元素，用 console.log 打印出图片的 URL（src 属性）到控制台。（提示：要想完成挑战，你应该注释掉 prompt.js 的最后一行，见清单 7.17。）

清单 7.17　注释掉 prompt.js 的最后一行

```
// 将输出插入到网页中
// document.body.innerHTML = output;
```

要完成这个挑战所需的所有知识你都已经学到了，所以我不会给你展示如何做。以清单 7.16 为例，如果你遇到困难，到网上搜索一下。如果你完全卡住了，可以参考本书网站上第 7 章内容中的解决方案。如果你靠自己完成了这个挑战，你会对这些概念有更好的理解，所以最好把网上发出来的解决方案当作最后不得已的一招。完成后，你应该可以在谷歌浏览器的开发者工具控制台中看到类似图 7.3 展示的内容。

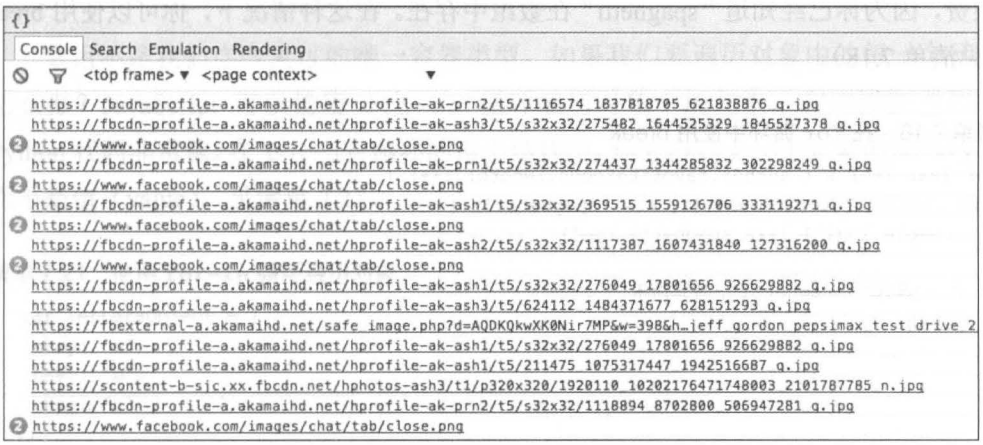

图 7.3 遍历所有图片会得到大量图片 URL

嵌套循环

`for` 循环的代码体可以包含任何代码，包括另一个 `for` 循环。嵌套 `for` 循环在谨慎使用的情况下会很有用，但是它们可能会显著降低程序速度。要了解原因，我们想象一下，你要遍历 Facebook 好友列表，对于每一位好友，要遍历他们的 Facebook 状态，寻找一个特定词汇（“我朋友中有没有人曾经发过 `for` 循环相关的内容”）。假设平均每个 Facebook 用户有 100 条更新，如果你只有一个好友，你只需要执行内部 `for` 循环 100 次，但如果你有两个好友，你需要执行内部 `for` 循环 200 次。内部循环的执行次数随着好友数的增加而快速增加。如果你有 10000 个活跃好友，你的程序会执行很久。如果你发现自己正在用一个嵌套 `for` 循环，要注意其成本会很快增加。有时候用另一种方式来组织数据会帮你缓解这个问题。

你需要停下来

在某些场景，你需要在条件变为假之前终止 for 循环。例如，假设你想要检测意大利面（spaghetti）是不是我最喜欢的食物之一。author.favoriteFoods 数组在我们的示例代码中很小，但我们假设它有几百个元素（这样可能更反映现实）。你可以遍历数组中的全部元素，检测是否每一个都是'spaghetti'。不过你不需要遍历全部元素。如果数组中的第二个元素就是'spaghetti'，遍历全部其他元素就是浪费，因为你已经知道“spaghetti”在数组中存在。在这种情况下，你可以使用 break（见清单 7.18）。

清单 7.18　在 for 循环中使用 break

```
for (var i=0; i < author.favoriteFoods.length; i++) {
  if (author.favoriteFoods[i] === 'spaghetti') {
    console.log('I like spaghetti too');

    // 一旦找到"spaghetti", 停止循环
    break;
  }
}
```

forEach

for 循环语法有可能会让人很困惑，特别是你要做的就是遍历数组中的每一个元素。不过，幸运的是，有更好的遍历数组的方法（只要你用的不是 IE9 之前版本的浏览器）。数组有内建的 forEach 方法做循环遍历（见清单 7.19）。如果你想在 getImages.js 文件中的 images 数组上使用 forEach，你会遇到问题。因为 images 实际上是一个类似数组的 HTML 元素集合，而不是真正的数组。因此，images 不支持全部数组方法，包括 forEach。在这一点上，JavaScript 有点奇怪。

清单 7.19　用 forEach 遍历数组

```
> var lettersArray = ['a', 'b', 'c'];
  ["a", "b", "c"]

> lettersArray.forEach( function(letter) {
    console.log(letter);
  } );
  "a" "b" "c"
```

很多语言都对遍历数组元素提供了方便的方法，因为这是一个常见任务。不同语言的语法略微不同，但思想都是一样的：对数组中的每一个元素，执行一些代码。Python 有一个简单的语法，很适合与 JavaScript 比较（见清单 7.20）。

清单 7.20 遍历列表（Python 中数组称为列表）

```
lettersList = ['a', 'b', 'c']
for letter in lettersList:
   print letter
```

数组不是我们迄今见过的唯一容器类型。如果我们要遍历对象中的每一个元素怎么办？JavaScript 可以使用 `for-in` 循环遍历对象中的元素，和你刚刚看到的 Python 代码非常像。在 `for-in` 循环的每个迭代中，你会得到下一个元素的键（key），而不是值（value）。见清单 7.21。

清单 7.21 使用 `for-in` 循环遍历对象

```
> var lettersObject = {
    aKey: 'a',
    bKey: 'b',
    cKey: 'c'
  };
  Object {aKey: "a", bKey: "b", cKey: "c"}
> for (key in lettersObject) {
    console.log('The key is: ' + key);

    // 如果你要得到当前元素的值，得用 lettersObject 对象
    console.log('And the value is: ' + lettersObject[key]);
}
The key is: aKey
And the value is: a
The key is: bKey
And the value is: b
The key is: cKey
And the value is: c
```

while

`while` 循环有点像 `if` 语句，也有点像 `for` 循环。`while` 像 `if` 语句是因为它也有一个条件，而 `while` 的代码体仅当条件为真时执行。`while` 循环像没有赋值和增量的 `for` 循环。代码体会一遍一遍的执行，直到条件为假。见清单 7.22。

清单 7.22　while 循环的基本语法

```
while ( /*条件*/ ) {

   /*主体*/

}
```

无限循环

for 循环仅仅执行有限次代码，因为增量操作会改变条件中的值，直到条件为假。while 循环没有增量操作，所以条件可能永远都不是假，而如果条件永远不是假，循环就会一直执行下去（理论上一大部分运行时环境不会让它发生）。（见清单 7.23）无限循环是不好的，你不想让相同的代码一直执行，对吗？

清单 7.23　无限循环

```
while (true) {

   // 真（true）永远不会是假（false），无限循环

}
```

要避免无限循环，你需要在循环的代码体中有指令修改条件值。假设你要循环遍历一个数组，每次遍历都将数组顶部的值取出来。如果你在循环体中修改了正在遍历的数组，for 循环会做些奇怪的事情，但 while 循环可以很好地处理这个问题。你可以使用数组长度作为条件，然后在循环体中取出一个数组的元素。

清单 7.24　使用 while 循环从数组顶部取出所有元素

```
var browserHistory = ['https://www.google.com',
                      'https://www.google.com/#q=droids',
                      'https://en.wikipedia.org/wiki/Star_Wars:_Droids'];
while (browserHistory.length) {
  console.log(browserHistory.pop());
}
```

当数组全部元素都被取出后，数组的长度会变成 0，而 0 是“假”，所以 while 循环结束。

再停一下

另一种避免无限循环的方式是使用 break 语句。while 循环中 break 语句的作用就像 for 循环中 break 语句一样。在大部分情况下，靠条件为假终止循环更好，但 break 语句也是一个不错的选择。

当你不知道什么时候停下

while 循环在你不知道循环需要执行多少次时特别有用。假设你在编写一个机器人程序，其工作就是清理孩子房间里的玩具（作为一个家长，我可以证明这样的机器人真的非常有用）。机器人的工作是一直不停地捡起玩具，直到没有玩具可捡。while 循环特别适合这种场景。机器人代码可能看起来如清单 7.25 所示。

清单 7.25 机器人不停地捡起玩具，直到没有玩具可捡

```
var thereAreToysOnTheGround = inspectRoomForToysOnTheGround();
var nearestToy;
while (thereAreToysOnTheGround) {
  nearestToy = locateNearestToy();
  pickUpToy(nearestToy);
  thereAreToysOnTheGround = inspectRoomForToysOnTheGround();
}
```

while 循环只要地上有玩具就会一直执行，甚至你的孩子在机器人收起玩具后又拿了一些出来，程序也会一直执行。while 循环很棒吧？

何时执行

用户界面最棒的功能之一就是可以根据我们的操作进行响应。当我点击邮件应用的“发送”按钮，邮件就被发出去了。当我旋转手机，屏幕的朝向就会翻转。有时，我们想让计算机在特定时间做些事情。我的日程表应用会在开会前 15 分钟给我发送提醒，我的电话在早上 5：00 播放闹铃，告诉我该起床了（好吧，我们可能并不希望电话把我们叫起来）。其他时候，我们想要计算机在一段时间之后执行一些操作。

事件

当某些事情发生时，执行代码的一种方式是使用事件。你可以把事件想象成一

个冰淇淋卡车的音乐。每个人都能听到音乐，但不是每个人都在意。如果你比较在意，你就会在听到冰淇淋卡车音乐时以某种方式进行响应（拿起你的钱包，跑向卡车，然后假装你在给一个孩子买冰淇淋，而不是你自己）。如果你不在意，就直接忽略音乐。

监听器

如果你的程序需要关注事件，你需要创建一个监听器（listener）。监听器主要由两部分组成：

1. 监听器关注的事件类型（见清单 7.26 中的`/* event type */`）。
2. 当事件发生时，监听器要做什么（见清单 7.26 中的 `function`）。

清单 7.26　JavaScript 事件监听语法

```
el.addEventListener(/*事件类型*/ 'click', function () {
  /*当事件触发时将被执行的指令*/
});
```

你可能想要监听用户点击按钮、按下键盘，或者点击一张图片（在移动应用中）等事件。每一种操作都会导致事件被触发。例如，我们给 `kittenbook.html` 创建一个事件监听器（见清单 7.27）。为了让我们的 JavaScript 更容易和 HTML 交互，我们要给包含问候语的 标签添加一个 ID，这样 Hello, World! 就变成 Hello, World 。我们要创建一个名为 `event.js` 的 JavaScript 文件，我们把事件监听器放在这里。记得要在 `kittenbook.html` 中引用这个新文件，就像 `kittenbook.js` 那样。

清单 7.27　添加一个点击事件监听器

```
// 使用我们刚刚加入到 kittenbook.html 中的 id 获取问候元素
var greeting = document.getElementById('greeting');

// 为问候元素的点击事件创建一个监听器
greeting.addEventListener('click', function() {
  if (greeting.innerHTML.match(/World/)) {
    greeting.innerHTML = 'Ola, mundo!';
  } else {
    greeting.innerHTML = 'Hello, World!';
  }
});
```

现在，在浏览器中打开 kittenbook.html 页面，尝试点击问候语。很酷吧？我们使用正则表达式和一个 if 语句来修改问候语，把它从英语变成葡萄牙语，再变回来。

定时任务

另一种基于时间执行代码的方式是使用定时任务。定时任务是指在特定时间执行的代码。最常见的定时任务是备份文件。你可以创建一个定时任务，将当天修改的文件全部备份，并设置成每天晚上 10 点执行。在设置了定时任务后，计算机永远不会忘记备份任务，你不需要一直记着它。创建定时任务超出了本书的范围，但如果你觉得有趣，快速搜索一下就可以很快上手。

超时

当你用 Gmail 发送邮件时，你会收到一个通知，说你的邮件已经发送，并给你一个选项来“撤销”这次发送。过一会儿，撤销选项就会消失，通知的内容变成你的消息已经发送。见图 7.4。

图 7.4 在一小会儿时间内，你可以撤销发送邮件，但之后就不行了

事实是，在邮件发送后，是不可能撤销的。Gmail 在撤销选项消失之前并没有真的发送邮件出去，如果你点了撤销，邮件就不会发送。JavaScript 有一个功能叫作 setTimeout，让你只能在给定的时间之后才能执行一些代码。如果 Gmail 使用 setTimeout，撤销功能的代码应该看起来类似清单 7.28。

清单 7.28 使用 setTimeout 撤销发送邮件

```
var emailForm = document.getElementById('email-form');

// 为邮件发送成功创建一个监听器
// 使用表单提交事件
emailForm.addEventListener('submit', function() {
  showUndoNotification();

  // 使用 setTimeout，在真正发送邮件之前等待 5 秒
```

```
// 5 秒就是 5000 毫秒，setTimeout 使用毫秒作为参数
setTimeout(function() {

  // 检测撤销按钮是否被点击
  if (undoWasNotClicked) {

    // 真正发送邮件并更新通知
    sendEmail();
    showMessageSentNotification();
  }
}, 5000);
// setTimeout 的第一个参数是我们要执行的代码，包装在一个函数中
// 第二个参数是在执行代码之前要等待多少毫秒
});
```

try

计算机程序是可以预期的，不过有时候会发生预期之外的事。计算机并不擅长处理意料之外的事情——你的手机应用会崩溃，文本处理器在你要保存的时候会自动退出，或者出现死亡蓝屏。程序可能直接退出，除非你告诉它们如何处理这些问题。处理问题的一种方法是 try/catch：尝试（try）执行一些可能会出问题的代码，然后接住（catch）这些问题，见清单 7.29。

清单 7.29 JavaScript 的 try/catch 语法

```
try {
  // 可能出错的代码
} catch (error) {
  // 如果在 try 块中发生错误，执行这里的代码
}
```

在事情出错前接住它

你可能感觉奇怪，你为什么一开始会想要写有风险的指令呢？是什么让指令变得有风险？还记得第 2 章“软件如何工作”我们写了一些伪代码来打开 Word 文档吗？我们看看清单 7.30 来回顾一下，这段代码其实非常危险。

清单 7.30 打开 Word 文档（第 2 章中的例子）

```
function openWordFile(filePath) {
  var wordWindow = openWordWindow();
  var fileContents = loadFile(filePath);
```

```
  displayFile(wordWindow, fileContents);
}
```

这段代码中比较危险的部分是 `loadFile(filePath)`。文件路径非法会发生什么？文件被删除会发生什么？文件路径指向了一个没有连接上的外置硬盘会怎么样？在加载文件时，很多事情都可能出错，但因为我们意识到那些代码有危险，我们可以把它们包在一个 `try/catch` 中，并优雅地处理错误。见清单 7.31。

清单 7.31 使用 try/catch 优雅地处理文件加载错误

```
function openWordFile(filePath) {
  var wordWindow = openWordWindow();

  // 尝试执行有风险的代码
  try {
    var fileContents = loadFile(filePath);
    displayFile(wordWindow, fileContents);
  } catch (error) {
    // 如果文件无法加载，显示一个合适的错误信息
    displayMessage('Sorry, the file ' + filePath + ' could not be loaded');
    displayMessage('due to the following error: ' + error);
  }
}
```

编写健壮的代码

在你写代码的时候，花些时间考虑下，哪些可能出错。在什么情况下你的代码会失败？如果代码失败，会如何影响用户体验？在大脑中带着这些问题，你可以使用像校验和 `try/catch` 等工具来写出能够处理各种疯狂事情的代码。这才是好的程序应有的代码，这才是你要写出来的代码。

总结

本章中你学到了很多如何写代码的知识。本章介绍的工具是一些对于写代码来说最重要的概念。你还写了遍历 Facebook 新鲜事里所有照片的代码，这是创建 kittenbook 扩展程序最重要的步骤之一。

本章你学到了：

- 操作符

- 流程控制
- `if` 语句
- `switch` 语句
- `for` 循环
- `while` 循环
- 事件、超时、定时任务
- `try/catch` 块

这里面每一个概念都用 JavaScript 代码做了示例。花些时间试验一下，这样你可以更好地理解如何使用它们。另一个可以更好地理解这些概念的方法是学习它们在其他非 JavaScript 语言中如何使用。在开始下一章之前，试着找出用 Python 或 Ruby 如何写出 `if`、`for` 及 `while`。有一个很好的在线资源可以尝试不同的语言：http://repl.it。在你学习每种语言的不同语法时，这些编程语言要素的本质就会变得更清晰。

下一章中，你会学到：

- 函数和方法
- 代码封装
- 代码重用
- 作用域

8

函数和方法

注

项目：将 Facebook 新鲜事中的照片替换成猫和狗的照片。

你的 kittenbook 代码还是一团糟。很抱歉这么说，但它确实是一团糟。全部变量都是全局变量（见后面关于“作用域”的章节），而且哪些指令用来执行哪些任务很不清晰。我认为这一团糟其实是我的错，因为这些代码的写法都是我告诉你的，尽管如此，它还是一团糟。你可能在质疑我的方法。如果我要教你如何编程，为什么我会告诉你写不好的烂代码呢？我保证有我的原因。首先，你不会明白简洁代码的价值，直到你写过一些烂代码。其次，要理解那些用来写整洁代码的工具，你需要有些烂代码才行。有些烂代码，总比一点代码都没写要好。

函数是最好的组织代码工具之一。一个函数是一组执行特定任务的指令，就像程序中的小程序。因语言而异，函数可能会被称为过程、子程序，或者方法。函数应该有一个单一任务，而且完成任务的全部指令都应该包含在函数之内。

函数结构

函数语法其实很简单（比 `for` 循环要简单得多），如清单 8.1 所示。每种语言的语法不一样，但基本上都一样。函数会用一个特殊关键字声明，比如 `function`、`def` 或 `sub`。函数会有一个名字，其应该描述函数执行的任务。声明函数的参数并且给出参数名。最后，函数有一组指令，称为函数体。如果函数返回一个值，函数体包含一个 `return` 语句。

清单 8.1　JavaScript 的函数语法

```
function countToTen( /*这里是参数*/ ) {
  /*代码在这里，在函数体中*/

  for (var i=1; i<=10; i++) {
    console.log(i);
  }
}
```

定义

清单 8.1 中的例子是一个函数定义。伴随着定义一个函数，你同时暗示了最终会在某个地方使用函数体中的代码，或者可能多次使用，但还没用到。当代码被执行，运行时会执行文件中的第 1 条指令，然后移动到第 2 条、第 3 条等。然而，当运行时遇到一个函数定义，它会注册这个函数，然后跳过函数体。要理解我的意思，试一下清单 8.2 中的代码。

清单 8.2　一个函数

```
console.log('Instruction 1');
console.log('Instruction 2');
function instructionThree () {
  console.log('Instruction 3');
}
console.log('Instruction 4');

/*
  输出:
  Instruction 1
  Instruction 2
  Instruction 4
*/
```

调用

函数体知道函数被调用才会执行。函数调用在大部分语言中都差不多：函数名字后面跟一对括号。括号里面可能有，也可能没有参数。要调用 countToTen 函数，你只需要使用 countToTen();。在控制台里试一下。你已经调用过很多函数：console.log('Instruction 1');是一个函数，phoneNumberPattern.exec

(phoneNumber);（在 prompt.js 里）也是。这些例子里的开闭括号之间包含参数（见下一节），但它们都是一样的函数调用。

参数

函数可以被设置成接收参数（在括号中间的那些东西）。参数是传入函数中的值，可以帮助明确函数行为。例如，console.log 的参数是你想要在控制台打印的文字。正则表达式的 exec 函数接受一个字符串参数，并尝试去寻找字符串中匹配正则表达式的结果。

参数可以让你的函数更灵活。例如，我们的 countToTen 例子一点都不灵活，假如我们只想要数到 7 怎么办？或者一直数到 15 呢？我们可以通过参数让 countToTen 数到任意数（当然我们最好把函数名字改了，因为它可以数到 10 以外的数）。见清单 8.3。

清单 8.3 数到任意数

```
function countTo(num) {
  /*代码在这里，在函数体中*/

  for (var i=1; i<=num; i++) {
    console.log(i);
  }
}

countTo(3);
```

形参

参数的概念和形参概念相关，而且这两个词通常可以互换。然而，两者之间有些细微的差别，而理解这些差别会对你理解函数有些帮助。形参是一种特殊的变量，在函数内创建，代表一个参数的值。在清单 8.3 中，num 变量是一个形参（function countTo(num)，而 3 是参数（countTo(3);）。函数在后台处理参数（3）赋值到形参（num）。如果你的函数接收多个参数，参数会按照出现的顺序赋值给形参。

清单 8.4 中，函数将参数 1 赋值给形参 a，将参数 2 赋值给形参 b，将参数 3 赋值给形参 c。

清单 8.4 多个参数

```
function manyParams(a, b, c) {
  console.log('a: ' + a);
  console.log('b: ' + b);
  console.log('c: ' + c);
}

manyParams(1, 2, 3);
```

参数过多

没有哪条法律规定一个函数可以有几个参数，但这并不意味着你可以写一个接受 200 个参数的函数。记住，参数会按顺序赋值给形参；如果你有 200 个参数，很容易把其中一个放错位置，而这个问题很难被发现。而且，如果你的函数接受过多的参数，这是一个很明确的迹象，表明这个函数做了太多事情。一条基本规则就是，函数不应该接受超过三个或四个参数；如果函数需要更多参数，很可能这个函数应该被拆分成多个函数。

return

第 2 章“软件如何工作”中，你学到了软件的输入和输出。对于函数来说，参数就是输入，而返回值就是一种输出。有些函数的目的是获取（get）一些东西，而不是做一些事。prompt 函数就是获取一些东西的例子（prompt 的输出是用户输入到弹出框中的内容）。现在 prompt.js 的第一行应该看起来更好理解了；我们调用 prompt 函数，然后将它的输出赋值给 userName。见清单 8.5。

清单 8.5

```
var userName = prompt('Hello, what\'s your name?');
```

对于一个有返回值的函数，函数体必须有一个 return 语句。如果函数体没有 return 语句，函数会返回 undefined（console.log 没有返回值，这就是为什么每次你运行 console.log 时，控制台会出现 undefined）。清单 8.6 展示了一个 sum 函数的例子，用到了 return 语句来返回两个数的和。作为练习，你可以试着写一个 product 函数，返回两个数的积。

清单 8.6 返回两个数的和

```
function sum (x, y) {
  return x + y;
}
```

return 语句通常是函数体中最后执行的一条指令，即使 return 语句后面还有其他指令。return 语句是用函数的方式在说："我做完了所有要做的事，这是结果。"你可以让 return 语句的"提前退出"功能为你所用。比如，你可以使用 return 语句替代 break 语句来退出循环。想想我们在第 7 章"何时使用 If、For、While"里用 for 循环来判定"spaghetti"是否在最喜欢的食物数组中。清单 8.7 展示了在函数的循环中，使用 return 替代 break 语句。

清单 8.7 用 return 提前退出循环

```
function doesStevenLikeSpaghetti(favoriteFoods) {
  for (var i=0; i < favoriteFoods.length; i++) {
    if (favoriteFoods[i] === 'spaghetti') {
      // 一旦"spaghetti"被找到，停止循环，返回真
    }
  }

  // 如果循环结束，意味着"spaghetti"没有被找到，那么返回假
  return false;
}
```

尝试重写清单 8.7 中的函数，让它可以检查任意食物，而不只是 spaghetti（意大利面）。要确保你给这个函数起一个能够反映其新功能的新名字。

调用栈

函数可以调用其他函数，这个函数又可以调用其他函数。哪个函数调用了哪个函数的列表被称为调用栈。当函数被调用，这个函数会被加到调用栈的顶部。当函数执行结束，这个函数会从调用栈中移除。清单 8.8 给出了函数调用函数的例子，而图 8.1 展示了调用栈在 third 被执行时的样子。

清单 8.8 函数调用函数

```
function first() {
  console.log('Executing first function');
  console.log('Calling second function');
```

```
  second();
}
function second() {
  console.log('Executing second function');
  console.log('Calling third function');
  third();
}
function third() {
  console.log('Executing third function');
}

first();
```

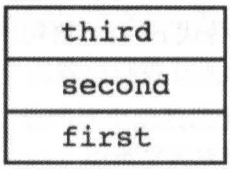

图 8.1　清单 8.8 的调用栈，当 third 被调用时的状态

代码封装

函数的一个主要作用就是组织你的代码。函数可以整齐地打包（或者封装）很多指令，这样它们就可以在一条指令中调用。从代码组织的角度来说，函数中的全部指令应该是相关的。

一次做好一件事

一个函数应该只有一件事要做，而且应该做好这一件事；每个需要做这件事的地方，都应该通过调用该函数的方式实现；而任何跟完成这件事无关的指令都不应该出现在函数中。无关的代码可能引起奇怪的副作用。清单 8.9 展示了这种奇怪行为可能会在函数包含无关代码时发生。

清单 8.9　铺床（以及其他）

```
function makeTheBed(bed) {
  // 把枕头放在床上
  pillowsOnBed = true;

  // 把床单放在床上
  sheetsOnBed = true;
```

```
  // 拉直床单
  sheetsStraightened = true;

  // 把毯子放到床上
  blanketsOnBed = true;

  // 拉直毯子
  bed.blanketsStraightened = true;

  // 刷牙，因为我总是在铺完床之后刷牙
}

// 我今天在探望奶奶
goToGrandmasHouse();

// 要帮奶奶做一些家务
makeTheBed(grandmasFeatherBed);

// 糟了，我刚刚用奶奶的牙刷刷了牙
```

brushTeeth 任务可能看起来和 makeTheBed 函数中的其他任务有关系，因为我通常在铺床之后去刷牙，但是拜托——我真的铺了床就一定要刷牙吗？清单 8.9 中，我去祖母家，然后铺了床，然后我刷了牙（用祖母的牙刷），因为 makeTheBed 函数表明了我必须在铺床任务结束后刷牙。诚然，这个例子有些极端，但这在组织代码时真的是一个问题。

> **函数命名**
>
> 如果你无法想出要给函数取个什么名字，这是一个迹象，表明你的函数可能没有明确的目的。你的函数本来要执行什么工作？这个问题的答案应该就是你的函数名。当你给函数找了个好名字，你可以很轻松的确定某个任务是否属于这个函数。makeTheBed 是一个很好的名字，因为函数的目的和范围都很明确。刷牙显然不属于铺床这个函数。

分而治之

在你开始思考软件需要执行哪项具体工作时，构建一整个计算机程序就变得不那么吓人了，而你的代码也会变得更干净，更有条理。顺着这条线，我们来想想，kittenbook 应该执行哪些工作：

- 获取用户名
- 获取用户电话号码
- 基于电话号码确定用户的位置
- 获取用户 Facebook 新鲜事中的图片列表
- 根据用户的位置，用猫猫狗狗的照片替换列表中的照片

现在，执行这些不同工作的代码都混杂在一起，很难看出哪段代码是做哪件事的。我们给每一项工作都创建一个函数。你的首要任务是给每一个函数取一个好名字，然后把代码分到各个函数中去。首先，你需要创建一个新的文件，名为 `main.js`，它会被用来当作程序的入口。在你创建好函数后，记得你必须调用它们，否则代码不会执行。

作为开始，清单 8.10 展示了新的 `main.js` 文件。清单 8.11 展示了 `prompt.js` 可能的样子，而清单 8.12 是新的 `replaceImages.js` 文件的代码。你还需要在 `getImages.js` 文件中创建一个 `getImages` 函数，在 `value.js` 文件中创建 `getAreaCodes` 函数。如果你卡住了，可以在本书的网站找到所有文件的示例代码。

清单 8.10 `main.js` kittenbook 控制中心

```
function main() {
  var userName = getUserName();
  var phoneNumber = getPhoneNumber(userName);
  var location = getLocation(phoneNumber);
  var images = getImages();

  // setInterval 就想 setTimeout
  // 除了它是重复执行，而不是执行一次
  // 用 setInterval 来替换那些随着滚动页面加载上来的新图片
  setInterval(function() {
    images = getImages();
    replaceImages(images, location);
  }, 3000);
}

main();
```

清单 8.11 封装后的 `prompt.js`

```
// 获取用户名
function getUserName() {
```

```
  var userName = prompt('Hello, what\'s your name?');

  if (!userName) {
    userName = prompt('You didn\'t enter a name. Really, what\'s your name?');
  }
  return userName;
}

// 获取用户电话号码
function getPhoneNumber(userName) {
  var phoneNumber = prompt('Hello ' + userName +', what\'s your phone number?');
  if (!validatePhoneNumber(phoneNumber)) {
    phoneNumber = prompt('Please enter a valid phone number.');
  }
  return phoneNumber;
}

// 验证电话号码的有效性
function validatePhoneNumber(phoneNumber) {
  return phoneNumber.match(/(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/);
}

// 基于电话号码确定位置
function getLocation(phoneNumber) {
  // 创建电话号码模式
  var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
  // 从phoneNumber 中获取匹配的部分
  var phoneMatches = phoneNumberPattern.exec(phoneNumber);
  var areaCodes, areaCode, locationName;
  if (phoneMatches) {
    areaCode = phoneMatches[1];
    areaCodes = getAreaCodes();
    locationName = areaCodes[areaCode];
  }

  // 看，这是个三元操作符
  // 如果存在，返回 locationName，否则返回"somewhere"（某地）
  return 'somewhere' return locationName ? locationName : 'somewhere';
}
```

替换 Facebook 新鲜事图片的代码，如清单 8.12 所示，可以分为 4 个部分。

1. 基于 `location` 参数，把图片按照替换的类别分开。`baseImageUrl` 变量会被用来获取图片。http://placepuppy.it/有一些小狗的图片，而 http://placekitten.com/g/有一些小猫的图片。你可以在这些 URL 的后面添加一些尺寸来获取特定尺寸的图片，

比如 http://placepuppy.it/300/500 会返回给你 300 像素宽、500 像素高的小狗图片。相比 placepuppy，我更喜欢 placekitten，因为 placekitten 更可靠，图片加载速度更快，但你可以两个都试试。

2. 循环遍历 images 参数的所有图片。

3. 在循环中得到每一张图片的尺寸，我们要让新图片跟原始图片完全一样大小。

4. 使用第一步中的 baseImageUrl 和第三步中的尺寸设置图片的 src 属性。

src（源，source 的缩写）属性告诉浏览器去哪里找图片。你在 Web 上看到的所有图片都有一个源，这些源都是 URL。我们更新了图片的源属性，就基本上是告诉浏览器在图片的位置展示一张新图片。

清单 8.12 用猫猫狗狗的照片替换全部图片

```
function getImageHeight(image) {
  return image.height;
}

function getImageWidth(image) {
  return image.width;
}

function replaceImages(images, location) {
  var baseImageUrl, height, width, image;
  switch (location) {
  case 'Memphis':
    // 如果是孟菲斯(Memphis),就用小狗的图片
    baseImageUrl = 'http://placepuppy.it/';
    break;
  default:
    // 其他地方用小猫的图片
    baseImageUrl = 'http://placekitten.com/g/';
    break;
  }
  for (var i=0,len=images.length; i<len; i++) {
    image = images[i];
    height = getImageHeight(image);
    width = getImageWidth(image);
    image.src = baseImageUrl + width + '/' + height;
  }
}
```

物尽其用

你可能在看 kittenbook 代码时注意到了，有些指令确实和我们列出来的工作都没关系，项目中的每一行代码应该都帮助我们的程序实现目标，这些不必要的代码在我们添加函数之前很难识别出来，但现在我们可以很清楚地看到，我们可以抛弃掉那些代码。删除没用代码的行为应该得到鼓励。如果你的程序中有代码没有帮助你达成目标，把它们都抛弃掉吧。

代码重用

迄今为止，我们使用了函数来组织代码，但函数还能做得更多。函数的好处之一，是它们可以被任意多次调用，从任何地方调用。

解决通用问题

编写可重用代码的一个关键点在于改变你解决问题的思考方式。不要只是想着解决你面前的问题，而是考虑要解决一类问题，然后建立一个解决方案，能够对所有同类问题都适用——而不仅仅是你面前的问题。改变思考方式不是一件容易的事，需要不断地实践和经验。从本章开始，我介绍了 `countToTen` 函数，其解决了特定的“从 1 数到 10”的问题。然后我们构造了 `countTo`，一个有参数的函数，能够从 1 数到任意数字，解决了一个更通用的问题。

用更少的代码做更多的事情

现在，假设我们要从 10 数到 15，我们不能用 `countToTen` 或者 `countTo`，因为它们都是从 1 开始。我们可以写一个新的函数，`countFromTenTo`，但还不够好。假如我们想要从 11 或者 9 开始数呢？我们真正想要的是一个函数，可以从任意数字开始，数到任意数字。我们把这个函数命名为 `countFromXtoY`，见清单 8.13。

清单 8.13 一个通用的从任意数字数到任意数字的函数

```
function countFromXtoY(x, y) {
  for (var i=x; i<=y; i++) {
    console.log(i);
  }
}
```

现在这个函数就可以重用了。我可以用 countFromXtoY(10, 15)来从 10 数到 15，或者用 countFromXtoY(300, 5000)来从 300 数到 5000，或者用 countFromXtoY(10, 1)来从 10 数到 1。等等！最后一个无法工作！试一下——然后你会看到 countFromXtoY 只对 x 大于 y 的情况适用。我们可以让函数真正的可重用和弹性，但代码会复杂一点。见清单 8.14。

清单 8.14 真正可重用的 countFromXtoY

```
function countFromXtoY(x, y) {
 if (x > y) {
  // 如果 x 大于 y，意味着我们要从大往小数

  var incrementor = function(idx) {
   // 将索引指向的值减 1
   return idx - 1;
  }
  var comparison = function(idx, num) {

   // 直到索引不再大于等于终止数字
   return idx >= num;
  }
 } else {

  // y 大于等于 x，我们要从小往大数
  incrementor = function(idx) {

   // 将索引指向的值加 1
   return idx + 1;
  }
  comparison = function(idx, num) {

   // 直到索引不再小于等于终止数字
   return idx <= num;
  }
 }

 // 起始数字是 x
 // 使用比较函数来确定我们是否要继续循环
 // 使用递增函数来确定朝哪个方向计数
 for (var i=x; comparison(i, y); i = incrementor(i)) {
  console.log(i);
 }
}
```

这个版本的 countFromXtoY 可以从任意数字到任意数字进行正数或者倒数（尽管可能有些让人迷惑）。上一章中，我们讲到 for 循环有一个赋值语句，一个比较语句和一个增量语句。清单 8.15 中的函数使用内部函数进行比较和增量，而这些函数的行为依赖于 x 是否大于 y。

赋值函数

清单 8.14 引入了函数的一个新特性：在 JavaScript 中，你可以把一个函数赋值给一个变量。在 countFromXtoY 中，我们必须将函数赋值给变量，而不是用正常方式定义函数，因为 JavaScript 有个叫作吊装（hoisting）的有趣特性。吊装超过了本书的范围，但如果你感兴趣的话，可以自己搜索一下。

不要做重复的事（DRY）

DRY 的概念在编程中多次出现（你已经在数据库设计中看到——不要在多个表中重复数据），在组织代码方面尤其重要。如果你有两个不同的函数，做的事情几乎一样，你可能要写两遍同样的指令，这是浪费时间。更不好的是，当你发现函数中有个 bug（相信我，你一定会找到 bug），你会不得不在两个函数中修复 bug。甚至更糟的是，当你在一个函数中修复了 bug，但忘记在另一个函数里修复它，你可能最后陷入沉思："什么情况？我已经把这个 bug 修复了啊！"

我做的第一个大项目是给我的大学做一个相对复杂的网站。网站最重要的功能之一是自动补全（见图 8.2，在我做这个网站的时候还不知道这叫作自动补全）。在建设网站的过程中，我可能写了 15 个补全函数，我不得不在每一个函数中重复大量代码。我做了很多复制粘贴的工作，每次我发现 bug，我不得不在 15 个不同的地方修改它。每次我发现一个更好的处理自动补全的方式时，我都必须更新 15 个函数。不要做重复的事——不然你就是在给自己增加工作。

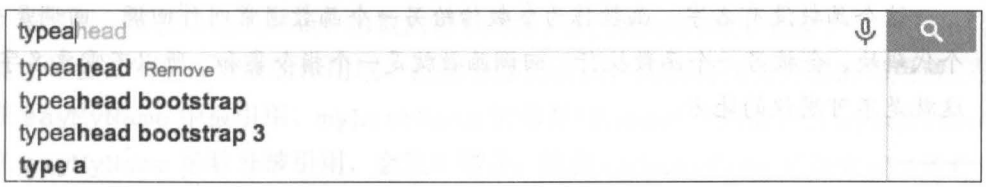

图 8.2 自动补全太有用了——如果你只需要写一次代码就更好了

我们回到数数的例子，如果我们做了很多从 1 数到 10，我们可能还是想要一个

countToTen 函数。但不要重复整个代码，我们可以在 countToTen 函数中使用 countFromXtoY 函数（见清单 8.15）。没有重复代码，太棒了！

清单 8.15　countToTen 使用 countFromXtoY

```
function countToTen() {
  countFromXtoY(1, 10);
}
```

匿名函数和回调

大部分函数都有名字。你在本书之前的代码示例中见过的每一个函数都有名字。你可以通过名字调用函数。但有些函数是匿名的；它们没有名字。如果你不知道名字，怎么调用函数呢？怎么会有人想要匿名函数呢？不可思议。好吧，我们回想下上一章 forEach 的例子。forEach 是一个特殊函数，用来访问数组的每一个元素，forEach 接受一个匿名函数作为参数，见清单 8.16 和 8.17。

清单 8.16　再现 forEach

```
> var lettersArray = ['a', 'b', 'c'];
  ["a", "b", "c"]
> lettersArray.forEach( function(letter) {
    console.log(letter);
  } );
  "a"
  "b"
  "c"
```

清单 8.17　forEach 的匿名函数

```
function(letter) {
  console.log(letter);
}
```

这个函数没有名字。函数作为参数传给另一个函数通常叫作回调。回调是一个代码块，会被另一个函数执行。回调函数就是一个指令集和，所以不需要名字。这就是不可思议的地方。

作用域

现在你应该对函数相当了解了，我们退一步，回想一下变量。变量是对某个值

的引用，可能是一个字符串，一个布尔值，或者一个对象。当程序运行过程中遇到变量，程序的运行时（Runtime）会根据变量名来查找对应的值。如果变量是在函数内定义，运行时就会只在函数的作用域内搜索变量名对应的值。在一个函数内定义的变量，不能在另一个函数中使用。这个概念有点绕，用一个例子就很容易明白了。见清单 8.18。

清单 8.18 变量和作用域

```
var myName = 'Steven';

function sayAName() {
  // 在函数中创建一个新的变量，
  // 使用和函数外面变量一样的名字（myName）
  var myName = 'Mike';
  console.log('myName inside sayAName is: ' + myName);
}

function sayMyName() {
  var myLastName = 'Foote';
  console.log('myName inside sayMyName is: ' + myName);
  console.log('myLastName inside sayMyName is: ' + myLastName);
}

sayAName();
sayMyName();
console.log('myName outside the functions is: ' + myName);
console.log('myLastName outside the functions is: ' + myLastName);
```

你一定要自己执行一下这段代码，亲自看看会发生什么。我们有两个相同名字的变量（myName），一个定义在所有函数之外，一个定义在函数内。myName 的值依赖于它在哪里被引用。当在函数外被引用，myName 的值是'Steven'，当在 sayAName 函数内被引用，myName 的值是'Mike'，因为 myName 在 sayAName 中被声明，等在 sayMyName 函数内被引用时，myName 的值又变成了'Steven'。运行时会首先查找函数的作用域，搜索变量的定义，然后再找更外层的作用域。当在 sayMyName 中被引用，myLastName 的值是'Foote'。然而，当 myLastName 在 sayMyName 函数外被引用，会发生错误，因为 myLastName 只在它定义的作用域中才存在。

全局变量

全局变量是指在所有地方都可用的变量。一般来说，全局变量就是在所有作用域之外声明的变量。全局变量可能会带来便利，因为它们在所有地方都可用。你不需要考虑作用域的问题，因为全局变量永远在作用域中。这真的很有用，但是全局变量也会带来问题。滥用全局变量会导致代码混乱，难以理解，并且难以调试。例如，清单 8.19 中的 sayAName 函数和清单 8.18 几乎一样，但这个很小的不同会导致很大的问题。

清单 8.19 小心使用全局变量

```
var myName = 'Steven';

function sayAName() {
 myName = 'Mike';
 console.log('myName inside of sayAName is: ' + myName);
}

sayAName();
console.log('myName outside of sayAName is: ' + myName);
```

不管在函数 sayAName 里面和外面，现在 myName 的值都是'Mike'，因为在清单 8.19 中，我们将 sayAName 函数中的 var 语句去掉了。var myName = 'Mike'; 是用来在函数 sayAName 作用域中声明新变量并对其赋值。但是 myName = 'Mike'; 只是一个给已存在于全局作用域的变量（全局变量 myName）进行赋值的语句。

全局变量很容易在代码的其他地方被覆盖。一个网页中的所有 JavaScript 文件共享全局变量（这也是在 kittenbook 的早期版本中，我们获取 kbValues 的方式）。你可能有一个全局变量，它的值对于成功执行程序非常重要，但这个变量的值很可能被其他文件的代码修改甚至删除，而你甚至对此一无所知。出于这些原因，我建议尽量避免使用全局变量，尽管它们可能带来便利。这样你的代码会更干净。这条建议不仅限于 JavaScript 作用域，这是条全局建议！

本地变量

本地变量是只在函数内可用的变量。当函数执行结束，本地变量就消失了。在函数内创建变量名时，你不用担心这些名字是否已在函数外部定义；函数内的作用域具有更高的优先级。不过，有些全局变量并不是你想要覆盖的。例如，清单 8.20

中的函数创建了一个本地变量，名为 console，然后想要使用 console.log，程序失败。原因在于，在函数的作用域内，console 的值是字符串'bash'，而字符串'bash'并没有 log 方法。

清单 8.20 小心选择本地变量名

```
function terminal() {
  var console = 'bash';
  console.log(console + ' is a cool tool to use in a terminal');
}

// 调用这个函数会产生一个错误
terminal();
```

变量查找是怎么工作的

当你在代码中使用一个变量，运行时必须找到这个变量的值。前面几个例子应该让你对这个过程有些了解了，但一个更直接的定义也许更有帮助。在查找变量的值时，函数首先在当前作用域中搜索变量，如果当前作用域中没有对应名字的变量，就会去定义了当前作用域的作用域中寻找。这一过程会一直重复，直到找到对应名字的变量，或者运行时到达全局作用域。如果对应名字的变量在全局作用域中都不存在，就会报错（会报引用错误，因为运行时没有变量的引用）。作用域的概念和变量查找曾经对我来说比难以理解；你可能会发现这一节值得多读几遍。为了避免像清单 8.20 里面的事情发生，你要理解作用域。

为了帮助更好的理解作用域和变量查找，将清单 8.21 中的代码表示在图 8.3 中。图中各个州的轮廓和美国的轮廓表示作用域。整个地球表示全局作用域。箭头代表变量查找发生的方向。箭头方向只有朝外的，因为外部作用域永远不能访问到内部作用域的变量（例如，全球作用域绝对不能访问定义在美国作用域的变量，而美国作用域决不能访问定义在北达科他州作用域的变量）。

清单 8.21 本地和全局作用域

```
var globalName = 'the world';
var name = globalName;

function USA() {
  var nationalName = 'United States of America';
```

```
  var name = nationalName;
  var capital = 'Washington D.C.';

  function nevada() {
    var name = 'Nevada';
    var capital = 'Carson City';
    console.log(capital + ' is the capital of ' + name + ' in ' + nationalName
            + ' in ' + globalName);
  }

  function northDakota() {
    var name = 'North Dakota';
    var capital = 'Bismark';
    console.log(capital + ' is the capital of ' + name + ' in ' + nationalName
            + ' in ' + globalName);
  }
  console.log(capital + ' is the capital of ' + name + ' in ' + globalName);
  nevada();
  northDakota();
}

USA();
```

图 8.3　变量只能朝外找，不能朝里面找

以作用域为目的的函数

如果你要让一个变量对所有函数可见，但不想引入全局变量的隐患，该怎么办呢？你可以创建一个函数，它的唯一功能就是限制作用域。你可以创建一个命

名函数，然后立即调用这个命名函数，如清单 8.22 所示，不过之后那个函数的名字就会出现在全局作用域中。更好的解决方案是创建一个匿名函数，并立即调用它，类似清单 8.23。JavaScript 中，这种方法叫作 IIFE（立即调用函数表达式），这个语法略显怪异。你要将一个匿名函数放在一个括号中，然后添加另一对括号来调用这个匿名函数。

清单 8.22 单纯用来限制作用域的命名函数

```
function restrictedScope() {
  var name = 'world';

  function greeting() {
    console.log('Hello, ' + name + '!');
  }

  function sayHello() {
    greeting();
  }

  sayHello();
}

// 好极了，但是 restrictedScope 现在是全局作用域中的一部分
restrictedScope();
```

清单 8.23

```
(function() {
  var name = 'world';

  function greeting() {
    console.log('Hello, ' + name + '!');
  }

  function sayHello() {
    greeting();
  }

  sayHello();
})();
```

总结

函数真的很棒，不是吗？你已经学了很多技巧让代码更有条理，更高效，更容易重用。本章你学到了：

- 函数的结构
- 如何用函数组织代码
- 用函数让代码更容易重用
- 作用域

下一章中，你会学到助你成为一个好程序员的技巧。具体来说，你会学到：

- 格式化代码
- 编写漂亮、可维护的代码
- 遵循编程最佳实践

9 编程标准

注

项目：创建一个.jshintrc 文件来检查 JavaScript 代码的质量。

到目前为止，前面的章节都偏向技术。学完了编程的方方面面，读写了很多代码，但编程还不仅是这些。你学了很多用来写程序的工具。在这一章，你要开始学习编写程序的规则，什么是好代码，以及如何写出让你自豪的代码。

“但是等下”，你会想，“我并不关心这些规则。只要我的代码能工作，我就满足了。”如果你只是想写点小程序，能帮你做些小事，还需要学习写高质量代码吗？好吧，是的。编程是一个强大的工具，你需要学习如何正确的使用它。最初你的程序可能很小，你可能是唯一的用户，但很快你就会添加新功能，你的小程序就会开始长大。很快你的朋友和同事开始想要使用它们。如果你学习了这些规则，并且从一开始就写出了高质量代码，那些新功能就会更容易添加进来，而且你的朋友找到的 bug 也更容易修复。

学习一门编程语言的规则就像掌握一种运动的技巧。当打篮球时，你可能很少在半场就进行超远投篮，但如果你每一次投篮都是在中场投出去，没人会想要和你一起玩了。类似的，你可以不按照规则、标准和惯例写代码，但没人会愿意和你一起工作（包括未来的你）。

编码惯例

不论你用什么编程语言，你都应该遵循某些标准。有些标准是帮你避免代码中的 bug。比如，JavaScript 中，你应该一直使用===（三等号）而不是==（双等号），因为==可能会得到意外结果（见第 7 章“何时使用 If、For、While”）。其他标准

就真的只是惯例，帮你保持代码一致性和可读性。有些惯例应该在使用一门特定语言的时候都遵循，比如，JavaScript 变量名应该用驼峰式（myName），而 Python 变量名应该使用全小写字母和下画线用来分隔单词（my_name）。下画线在 JavaScript 中可以用，驼峰也可以用在 Python 中，但实际上每一个用这些编程语言的人都会遵循惯例。

有些惯例根据项目不同而不同，甚至在同样的编程语言中。比如，有些 JavaScript 项目使用单引号（'a single quoted string'），而其他地方使用双引号（"a double quoted string"）。这些惯例只是一种倾向性，但每一个工作在这个项目中的人都应该遵守。

设定标准

说到遵循标准和惯例，你应该确保你熟悉这门语言和项目中的项目标准和惯例。如果你想要开始在一个 Python 项目中工作，快速在搜索引擎中搜索类似“Python coding standards”这样的东西，你应该会找到一些关于 Python 标准的资源，然后为项目设置自己的标准。你可能现在还不知道要设置什么标准，没关系。如果你找到了语言的通用标准，你就比多年前的我做得更好了。你可以仅仅以一条标准开始：“避免使用全局变量”。随着你不断地编程，你会找到一些总是出问题的小事情。将它们作为你的标准的一部分记下来，提醒你不要那么做。你应该总是会对半年前自己写的代码多少感觉有些羞愧。这证明你在进步。

你已经有了一个项目，所以你可以尝试现在设置标准。还记得第 4 章“构建工具”中学到的构建工具吗？还记得我们是怎么设置 JSHint 来检查 JavaScript 代码质量的吗？你已经在第一个项目中有了编程标准，而这些编程标准可以通过 JSHint 和 Grunt 来通过编程方式强制遵守。

黑科技，用还是不用

写高质量代码是为了你自己好。以后，你会学到更多，你的代码也会运行得更好，也更容易修复 bug。不过，有时并不必要或者说不需要一个高质量的长期解决方案。你要形成这样的能力，判断什么时候一个使用黑科技的短期方案要比长期健壮的方案更好。

黑科技（hacking）

可能别人跟你说过，黑客行为是指获取安全系统权限："黑进银行系统，偷一大笔钱。"可能在大众文化中，这就是黑科技的意思。在程序员中，黑科技通常不带有贬义。很多程序员自称黑客，很多知名公司会定期举办名为"黑客马拉松（hack-a-thons）"的活动，让"黑客"们聚在一起，搭建很酷的新应用和服务。在这些例子中，黑科技通常是指写代码的行为，它是好事。

黑科技还可以表示写出数字世界的强力胶带和铁丝一样的代码。解决方案可能很丑、脆弱、可笑，但可以工作。我再说"黑科技，用还是不用"时，指的就是这种黑科技。

立即付款还是先用后付款

如果你现在写一些丑陋但能运行的代码，在未来，你就会为 bug 和维护付出代价。通常值得现在花些时间做些工作，因为相比一开始写好代码，修 bug 花费的成本可能要多很多。然而，在某些场景下，快速写出丑陋但可用的代码其实是对的。如果你正在为了学一个新概念而写一些代码（比如，在 console 里做实验），那就没有理由遵循所有规范。代码执行完就马上被抛弃了。如果你想要快速构建一个原型来证明某个想法可以实现，一个快速、丑陋但可用的解决方案可能是最合适的。当你要决定是写快速但丑陋的代码，还是写高质量代码时，你应该明白你是在决定要现在付出代价，还是未来付出代价。决定要未来付出代价，通常叫作技术债，这个概念就像信用卡透支。

写可维护的代码

创建并遵循编码规范的最大动力之一，就和软件开发生命周期相关。软件开发生命周期基本上包含 4 个部分。

1. 有一个想法。
2. 计划如何将想法变成现实。
3. 写代码将你的想法变成现实。
4. 通过修复 bug、添加新功能（维护）来保持"想法变成现实"的过程一直持续。

这 4 个步骤不是都需要一样的时间。那个想法可能只是你洗澡时候 5 秒钟想出

来的。计划如何实现想法应该会比 5 秒钟长很多，但我们常常太兴奋，急于马上开始第三步，以至于我们跳过了第二步（羞愧）。将想法变成现实的过程会花更长时间。依赖于想法的宏大程度，可能要花几天、几个月甚至几年。然而，到目前为止最长的一步，是维护。只要这个项目还在，你就要持续维护它。如果你写的代码质量高，可维护性好，维护阶段也可能会很愉快。如果你选择不设置编码规范，也不写可维护代码，维护阶段就会很痛苦。

什么才是可维护代码？这个（不完全）列表应该能说明一些基本思想：

- 文档完善（好的注释，图表等）
- 提前想好并做好计划（你是否在一开始就把事情想清楚，还是直接就开始写代码？）
- 完善的测试，对 bug 相对免疫（你是否完整测试了程序，还是只是猜测它能运行？）
- 容易理解（对那些了解编程语言的人来说）
- 可重用（见第 8 章“函数和方法”）

代码格式化

编写好代码要比仅仅将代码按照正确顺序写出来更重要。代码的颜值会对程序的成功有着巨大的影响。清单 9.1 展示了超简洁格式的 `values.js`，清单 9.2 展示的 `values.js` 稍好但不一致的格式化，清单 9.3 展示了正确格式化的 `values.js`。哪一个更容易理解？哪一个更好用？

清单 9.1　超简洁格式化的 `values.js`

```
var kbValues = {projectName:'kittenbook',versionNumber:'0.0.1',areaCodes:{ '408':'Silicon
Valley','702':'Las Vegas','801':'Utah','765':'West Lafayette', '901':'Memphis'}};function
getAreaCodes() {return kbValues.areaCodes;}
```

这个看起来极其让人困惑的代码，其实和清单 9.2 和 9.3 展示的代码是一样工作的。区别在于，要读懂清单 9.1 中的代码，你必须很有耐心，并能够忍受痛苦。

清单 9.2　不一致格式化的 `values.js`

```
var kbValues =          {
  projectName: 'kittenbook',
```

```
versionNumber: '0.0.1',
 areaCodes:{
   '408': 'Silicon Valley',
 "702":        'Las Vegas',
   '801': 'Utah'      ,
   '765':"West Lafayette",
   '901':      'Memphis'
}
};

function getAreaCodes() {
 return kbValues.areaCodes;
 }
```

我真的对于写成这样的代码感到反胃，得鼓起勇气才能看它一眼。我想我已经表达了我的观点，可以继续下一步了，因为我觉得我无法回过去看上面的代码。

清单 9.3 正确格式化的 values.js

```
var kbValues = {
 projectName: 'kittenbook',
 versionNumber: '0.0.1',
 areaCodes: {
   '408': 'Silicon Valley',
   '702': 'Las Vegas',
   '801': 'Utah',
   '765': 'West Lafayette',
   '901': 'Memphis'
 }
};

function getAreaCodes() {
 return kbValues.areaCodes;
}
```

> **计算机不理解编程语言**
>
> 在你考虑如何格式化代码时，记住你的计算机无论如何都不理解你的源代码。源代码必须在计算机理解之前先转换成二进制文件。源代码是给人读的，所以格式化应该让人更容易理解。

保持一致

格式化代码的方式由你自己决定（或者和你一起工作的团队）。如你刚刚看到的

三段代码，你格式化代码的方式不影响代码运行（作为警告，看下“Python 风格”）。你应该在决定如何格式化代码之后，贯彻遵循你所选的格式化方式。保持一致的格式化还可以帮助你避免 bug。在本书中的所有示例中，我都在字符串中使用单引号，但我本可以使用双引号的。JavaScript 都能理解。然而，如果你在字符串的开始使用双引号，就要以双引号结尾，以此类推。如果你在某些字符串中使用双引号，而在另一些中用单引号，你可能会不小心在字符串的开始使用双引号。这就无法工作，JavaScript 会被搞糊涂，而你的代码也被搞坏了。

空白字符

在用于编码的代码格式化规范中，一致使用空白字符可能是最重要的。适当的使地空白字符会极大地提升代码可读性。再想想清单 9.1 和清单 9.3 的区别。完全是一样的代码，但是清单 9.1 移除了全部空格和换行符。使用空白字符的指导原则是，空白字符应该和你的代码逻辑保持一致。函数中的代码行应该用同样的缩进，`for` 循环中的代码行也要有同样的缩进。这样，哪些行会在 `for` 循环还是函数中执行就会很清晰。清单 9.2 展示了使用不一致的代码缩进，会让读代码变得困难。

要使用缩进时，你可以用空格或者制表符，这个决定目前对你来说可能并不那么重要，但在程序员中这是个热议的话题（认真的）。说实话，我不觉得这两者之间有多大区别，但我更喜欢空格。大部分文本编辑器允许你开启软制表符，即在你按下制表键时，编辑器会帮你插入空格。不论你选择那一个，保持一致。混用空格和制表符，对于代码格式化不一致性来说，是最严重的问题之一。

> ### Python 风格
>
> 一致的格式有可能是个大问题。Python 语言尝试通过强制使用某些空白字符规范来避免这一问题。Python 使用缩进来表明哪些指令属于同一个代码块，而不是花括号。有些程序员非常愿意给空白字符赋予意义，就像 Python。不过，不管喜欢还是讨厌，你不得不承认，这么做确实提高了代码格式一致性。

清单 9.4　Python 的 `for` 循环

```
print 'Starting the for loop'

for i in range(10):
   print(i)
```

```
print 'Ending the for loop'
```

规则不会自己出现：要制定规则

在你确定了标准和规范之后，你一定要把它写下来。当你写下规则，你就更可能遵守它们。如果你在一个项目上和其他人一起工作，要达成一致，写下来的标准会帮助你们的源代码保持干净和一致，并且帮助你们避免冲突（“双引号更好！”，‘不，单引号更好！’）。这些写下来的标准有时候叫作风格设定。风格设定的形式之一就是JSHint 的配置文件，.jshintrc（清单 9.5 中的示例.jshintrc 应该被保存在kittenbook 目录下）。配置文件是我们在第 3 章“认识你的计算机”所讨论的那些隐藏的“点文件”之一。.jshintrc 的内容是 JSON 对象形式的一组配置，要了解更多可用的配置，请访问 http://jshint.com/docs/options/。

清单 9.5 kittenbook 的.jshintrc

```
{
  "predef": {
    "kittenbook": true,
    "kbValues": true,
    "prompt": true
  },

  "bitwise": true,
  "camelcase": true,
  "curly": true,
  "eqeqeq": true,
  "immed": true,
  "indent": 2,
  "latedef": true,
  "quotmark": "single",
  "undef": true,
  "unused": true,
  "strict": false,
  "trailing": true,

  "browser": true
}
```

我们.jshintrc 文件中最重要的几个配置（就我们讨论的这个话题来说）是

“predef”，在变量列表中的全局作用域里；"indent"，表示应该是用 2 个空格表示缩进；"quotmark"，表示我们会使用单引号；以及"browser"，告诉 JSHint 会看到像 document 这样的东西，因为我们的代码要在浏览器中运行（而不是 Node.js）。如果你想学习下其他配置的用法，可以看下前面提到的 JSHint 网站。

要让新的.jshintrc 配置工作，你还需要更新 Gruntfile.js，告诉 Grunt 在运行 jshint 时应该到哪里去找.jshintrc（见清单 9.6）。你可能还想要在 jshint 任务前运行 concat 任务，然后只检查连接过的文件。不管怎么说，连结过的文件才是我们真正想要检查的文件。我们还要更新 watch，因为之前它使用<%= jshint.files %>得到要监听的文件位置。最后，交换'concat'和'jshint'在 grunt.registerTask 中的顺序，这样 release/main.js 会在 jshint 任务执行前更新。

清单 9.6　为了使用.jshintrc 而对 Gruntfile.js 做的修改

```
module.exports = function(grunt) {
  grunt.initConfig({

    ...

    jshint: {
      options: {
        jshintrc: '.jshintrc'
      },
      files: ['release/main.js']
    },
    watch: {
      files: ['js/*.js', 'manifest.json'],
      tasks: ['default']
    }
  });

  ...

  // 注册任务
  grunt.registerTask('default', ['concat', 'jshint', 'copy']);
};
```

现在配置好了让 Grunt 使用.jshintrc，打开命令行，导航到 kittenbook 项目目录，然后执行 grunt，你应该会看到一些来自 JSHint 的错误。看看你是否清楚那

些错误是什么意思，该如何修复它们。如果你卡住了，可以在本书网站查看第 9 章的代码。

使用其他人的成果

我希望当年可以更早知道如何利用其他人的成果。我喜欢搭建东西，所以我故意无视其他人创建出来而我本可以重用的那些免费软件。这真是个糟糕的主意。尝试自己搭建所有东西，给了我很多教训。幸运的是，我明白了其他人也在做着很棒的工作。虽说如此，在自己搭建东西和免费地使用开源项目之间，存在一种平衡。

更快地构建

没有重用他人成果导致最严重后果的一个例子，可能是我坚持自己搭建一个自动补全系统时，我不愿意相信一个自动补全系统可以做得足够可重用，能够让我在自己的网站上使用其他人的自动补全系统（我拒绝相信的部分原因是我不理解回调）。我甚至不相信我可以做出一个可以用在自己网站不同部分的自动补全系统。我搭建了 15 个自动补全系统，简直要哭了，太浪费时间了。在之后的那个项目，我发现了一个开源的自动补全项目，我可以轻松将其插入我需要的地方。使用开源软件让我节省了大量时间。

不管是开源社区某个人的成果，还是团队另一个成员的成果，使用他人成果确实可以减少创建项目的时间。记住，我项目的目的不是构建一个自动补全系统，而是构建一个具有特定功能的应用，只是其包含了一个自动补全系统。包含一个其他人做好的自动补全系统不会降低我工作的价值，或是整个软件的价值。相反，我可以花更多时间优化我的应用。

开源软件

你可能听过或者读过开源软件相关的事，甚至那些头发不多的老板们都对开源软件有些印象：这是一个神奇的地方，可以解决你的所有问题，还不要钱！或者说，这有免费演讲和免费员工，他们聚在一起的希望是让世界变得更好。太棒了！

大部分商业软件都是分发打好包的、编译好的代码，但源代码是保密的。开源软件就是那些源代码是公开可获得的软件。Mozilla 的火狐浏览器（Firefox）、谷歌浏览器、安卓操作系统、Linux、JavaScript，以及 Python 全都是大型开源软件项目。如

果你有时间和耐心，你可以深入到安卓系统的源代码中，了解安卓的通知如何工作，因为它的源代码是公开的。

有一种观念是，任何人都可以给开源软件项目做贡献，每个人也可以以任何目的使用开源软件。实际上，开源软件有不同级别的自由程度。安卓是在谷歌内部开发的，源代码只有在新版本准备好后才被公开出来（封闭开发，开放源码）。大部分开源软件会包含某种具有以下两个基本目的的许可证：

1．你可以自由地使用和分发该软件。

2．如果你修改了软件，那些修改也要被公开。

这类许可证确保如果任何人或组织改进了软件，那些改进也要提供给每个人。这么说来，开源软件更像是维基百科：一个活跃项目，质量随着时间的推移而改进。

由社区建立

使用开源软件的最大好处之一，是每个人都可以贡献。如果你想要使用某个开源软件能够正确为德国用户格式化日期，你可以基本确定，一个了解很多日期格式化知识的德国程序员给这个项目贡献过代码。程序员会为了他们最擅长的领域而被聚在一起，所以开源社区通常都是正确的人在做正确的事。这些专家见过并且修复过与他们构建的软件相关的奇怪 bug，所以你不必具备这样的专业知识。你不需要理解解决方案中的全部错综复杂的事情，因为开源社区的专家程序员已经帮你做了这些事。

然而，你应该意识到，任何人都可以创建并发布一个开源软件项目。你找到了一个看起来能解决你的问题的开源软件项目，并不意味着你应该使用它。做些调研，其他人是不是在用？多少用户？有人活跃地在这个项目工作吗？是否有很多报告出来的 bug？使用低质量开源软件要比自己从头建的成本还要高。

什么时候该自己写

有时你的最佳选择是忽略开源选项，自己搭建一些东西。即使你发现开源项目看起来解决了你的问题，自己写还是有理由的。可能它没有完全解决你想要解决的问题。可能软件许可证让你（或者你的法律部门）不舒服。可能你对代码的安全性有些担心。可能你就是觉得自己做得更好。仅仅因为有些人已经搭建好了，并不意味着你就总得使用它。你会发现很多网络浏览器，很多操作系统，很多编程语言。

技术的进步，来源于相同问题，不同解决方案之间的竞争和互相学习。

最佳实践

某些编程标准超出了实际写代码的范围。我说的这些标准是文档、计划和测试。这些最佳实践会帮助你变成一个写出伟大程序的伟大程序员。随着你学会高效地使用这三个工具，你对程序的信心会不断提升。当我构建我的第一个网站时，我做了很少的计划和测试，只是偶尔写一点文档。结果，我对这个网站几乎没什么信心。每次做修改，不管大小，都会很紧张。新代码会不会把全部程序都破坏了？如果网站崩溃，我能修复好吗？最后我的恐惧被证实。我经常把网站搞坏，有时我好几天都无法搞清楚如何修复（我的用户对此很不高兴）。使用这些最佳实践会帮你避免我经历过的恐惧和痛苦。

文档

文档如此重要，以至于下一章的主题就是它。本书到现在为止，你只看到过一种文档，就是注释。对于你对代码的信心来说，文档至关重要。如果你没有在写代码的时候留下文档，你可能会忘记你的代码本来要做什么。尽管代码的目的是为了让人能理解，也并不意味着它就要写成像小说一样。理论上，你可以通读代码来了解它是在做什么，但那可能要花很长时间——而且你仍然有可能忘记当初为什么做了某个决定。如果你在做出决定的时候就将其记录成文档，就会更有自信，相信自己做了正确的决定。文档还有很多其他相关的内容，下一章你就会看到。

计划

计划的重要性被严重低估了。程序员想到一个想法就马上动手写代码，期望所有事情都会按部就班的做好（愧疚），这种事屡见不鲜。做计划让你成为更有信心的程序员，因为你会花些时间，通盘考虑整个问题，找到你认为的最佳方案。在你开始写代码时，你可能会发现你的计划有些偏离。在执行的过程中调整计划是完全可行的。我们会在第 11 章“计划”中详细讨论计划。

测试

在所有工具中，测试很可能会给你最多自信，让你觉得你写出了好代码。测试

让你知道，你的代码会在特定环境下，按照期望的方式执行。在第 12 章“测试和调试”中，你会学到如何写出可以测试你的代码的代码。每次你对代码作出修改，都可以运行测试，确保你的代码仍然如预期一样运行。在我搭建网站的时候，我真应该像上面说的那样写些测试。活到老，学到老，不是吗？

总结

本章没怎么讲写代码的纯技术细节（只有 6 个代码示例），更多的是关于如何写出更好的代码。我们讲了不少理论，但通过给 kittenbook 加入`.jshintrc`，我们将理论融入实践。本章你学到了：

- 编码标准
- 代码格式化
- 开源软件
- 编程最佳实践

下一章中，你会学到：

- 给软件写文档的不同方式
- 如何写出好文档
- 如何为自己写文档
- 如何为他人写文档

10 文档

注

项目：使用 JSDoc 和 Grunt 为 kittenbook 项目创建文档。

我知道你现在一定在想："真的有必要让我读一整章关于如何写文档的内容吗？如果有什么比写文档更无聊的事，那一定是读文档；如果有什么比读文档还无聊，那一定是读关于如何写文档的文档。我已经知道如何在代码中写注释了，还能有多难？"尽管我理解你的心情，但我希望你很快就会发现，文档对于编程并不像数码相机的说明书。编程文档实际上非常酷。你可以在你做一件事的时候，就在那里写下记录。想想一下假如莱昂纳多·达芬奇可以直接在他的作品上记录关于作品的笔记。好的文档，会让代码不那么神秘。

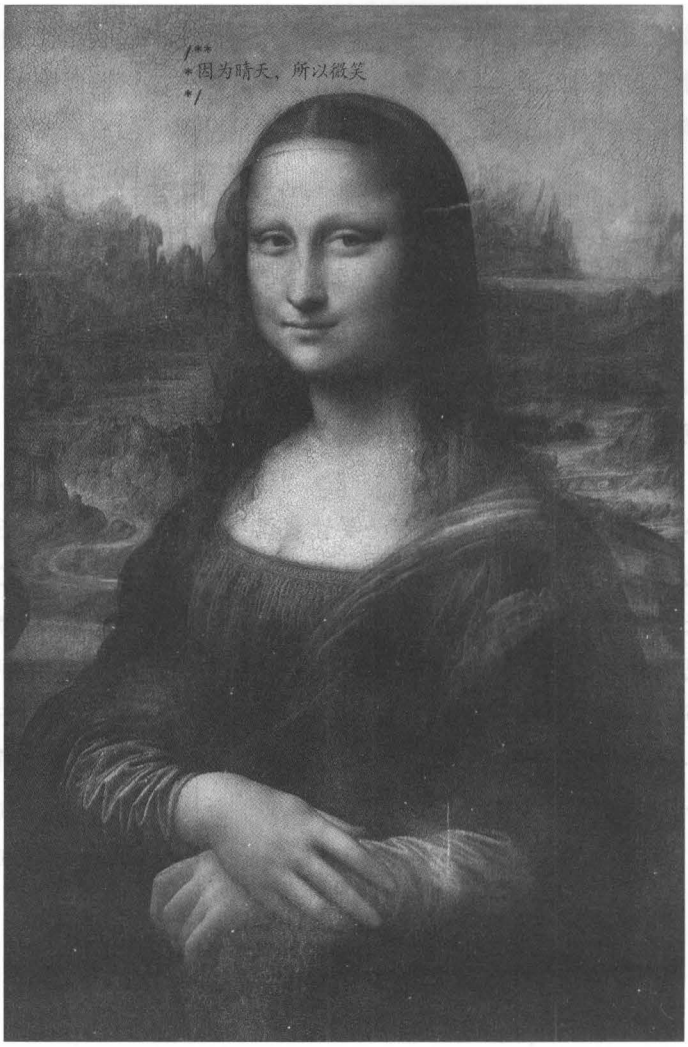

图 10.1 不再神秘，注释太棒了

文档化意图

为代码写文档总是好事情，但是不好的文档弊大于利。含糊、误导、错误及过时的文档会导致疑惑和挫折。写好文档不是一件容易的事情，因为你不仅要解释你的代码做了什么（这本身就很有挑战），还要确保简洁，并且没有过时。一个最好的方法是文档化代码的意图，而不是文档化代码实际做的事情。

自文档代码

不写坏文档的一种解决方案是，完全不写文档。欧耶！不写文档！先别激动，听我解释。现代编程语言的描述能力都很强，也就是说，代码可以读起来就像纯英语一样。如果你的代码读起来像英文，那还需要英文的注释干什么？不需要！我本人不是特别认同这个观点，但有些其他程序员相信，描述性的、自解释的代码已足够，没必要写注释。他们甚至认为这比有注释的代码还要好；如果代码就是文档，文档就永不过时。这一论断有很大价值，自文档代码背后的原则应当被接受，即使你还是选择写注释。

写出自文档代码最重要的一点，就是给变量和函数起具有描述性的名字。清单 10.1 是一个极端例子，展示了不好的变量名和函数名。清单 10.2 展示了相同的代码，但是用了具有描述性的名字。

清单 10.1 代码并不复杂，但相当难理解

```
function gL(pN) {
  var pNP = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
  var pM = pNP.exec(pN);
  var aCs, aC, lN;
  if (pM) {
    aC = pM[1]; aCs = gAC(); lN = aCs[aC];
  }
  return lN ? lN : 'somewhere';
}
```

清单 10.2 和 10.1 一样的代码，但是函数和变量名都是描述性的

```
function getLocation(phoneNumber) {
  var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
  var phoneMatches = phoneNumberPattern.exec(phoneNumber);
  var areaCodes, areaCode, locationName;
  if (phoneMatches) {
    areaCode = phoneMatches[1];
    areaCodes = getAreaCodes() locationName = areaCodes[areaCode];
  }
  return locationName ? locationName : 'somewhere';
}
```

你可能认识这段代码，就是 `prompt.js` 里面的 `getLocation` 函数。清单 10.1 中的代码完全不可读，但是清单 10.2 的代码相对容易理解。我在清单 10.2 中移除了

getLocation 的注释，但它仍然很容易理解（现在你理解了一些 JavaScript 代码之后就更好理解了）。当你写出像这样的描述性代码时，基本上就没必要写注释了。你甚至可以不用给正则表达式添加注释，因为 phoneNumberPattern 准确地描述了正则表达式的匹配意图。

不要耍花招

和大多数编程语言一样，JavaScript 有很多小把戏和捷径。有些很有趣，也很有用（见清单 10.14，这是我最喜欢的 JavaScript 把戏之一），但有些会让人迷惑。清单 10.3 耍了一个花招，检测一个元素是否在数组中。清单 10.4 展示了一个描述性的、可理解的方式，实现同样的功能。清单 10.3 中的~符号是位运算，表示取反，这部分超出了本书范围。我唯一见过将位运算取反用在 JavaScript 中的地方就是检查元素是否在数组中，或者一个字符是否在一个字符串中。indexOf 方法返回元素的位置，如果元素在数组或者字符串中不存在，返回-1. 所以检测元素是否在数组或者字符串中的方式之一，就是检测 indexOf 返回的数字是不是大于-1。大部分 JavaScript 开发者并不真正理解~符号的作用，所以清单 10.3 中的代码就属于耍花招。如果你的代码中花招太多，以至于需要添加注释来解释花招的作用，你很可能需要重写代码，让它更容易理解。

清单 10.3　使用~检查一个元素是否在数组中

```
var name = 'steven';
var searchForLetter = 'p';

if (~name.indexOf(searchForLetter)) {
  console.log('Apparently there is a "p" in "Steven"');
} else {
  console.log('The name is "Steven", not "Stephen"')
}
```

清单 10.4　使用容易理解的>方法检测元素是否在数组中

```
var name = 'steven';
var searchForLetter = 'p';

if (name.indexOf(searchForLetter) > -1) {
  console.log('Apparently there is a "p" in "Steven"');
} else {
  console.log('The name is "Steven", not "Stephen"')
}
```

不要将显而易见的东西写入文档

在第 8 章“函数和方法”中，你学到了 DRY 原则。“不要做重复的事情”的概念也要扩展到注释。把代码已经表达出来的意思再重申一遍没有任何价值。相反，这样的注释会破坏代码的流畅性，而且很快就会过时。清单 10.5 和清单 10.2 是一样的代码，但是清单 10.5 的每一样都写了注释。哪一个更容易读呢？

清单 10.5 用注释来重复一遍代码没什么用

```
/**
 * 获取位置
 */
function getLocation(phoneNumber) {
  // 创建电话号码模式
  var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;

  // 使用 phoneNumberPattern 得到 phoneMatches
  var phoneMatches = phoneNumberPattern.exec(phoneNumber);

  // 声明 areaCodes, areaCode, 以及 locationName 变量
  var areaCodes, areaCode, locationName;

  // 如果 phoneMatches 有效
  if (phoneMatches) {
    // 用 phoneMatches 数组中的第一个元素来设置 areaCode 变量
    areaCode = phoneMatches[1];

    // 使用 getAreaCodes 函数设置区域编码
    areaCodes = getAreaCodes();

    // 用 areaCode 和 areaCodes 设置 locationName
    locationName = areaCodes[areaCode];
  }

  // 返回 locationName 或者 'somewhere'
  return locationName ? locationName : 'somewhere';
}
```

函数名和变量名的描述性已经足够，这些额外的解释完全没必要。事实上，它们是有害的。存在这么多注释，哪一个才是真正的代码呢？真正的代码流程被没用的注释破坏了，这让清单 10.5 中的代码更加难以读懂。注释还可以扮演代码指示牌的角色，标记那些需要特殊注意，或者可能需要花些时间理解的代码。注释可以是

一个很好的指示，让程序员慢下来，然后仔细读这一段代码。如果注释到处都是，程序员在读代码时会不知道哪些代码需要额外注意。注释虽好，废话无用。

过时文档的危险性

只要注释是最新的就好，不过过时的注释会导致很多困扰。当注释与其标注的代码不同步时，它就过时了。到底注释是对的，还是代码是对的呢？这让人很困惑。而如果太多注释都是过时而且错误的，它们就会被忽略。过时的注释可能相当危险。任何时候你更新了代码，都应该记得更新注释。更新注释很容易被忘记，因为代码还能正确运行，尽管注释是不准确的。正因为你有时候会忘记，记录你的意图就很重要。即使底层代码变化了，你的意图一般不会变。

清单 10.6 中的代码和注释是同步的。然而，当我们决定在清单 10.7 中重构 `getLocation`（让它只做一件事，并且做好这件事），注释就变得不合理了。事实上，注释会变得有些误导。如果某人刚好读了注释，没有读代码，他会认为调用 `getLocation` 时不需要参数，但事实已经不是这样了。

清单 10.6　重构前的 `getLocation`

```
/**
 * getLocation 请求用户输入一个电话号码，
 * 然后基于电话号码返回一个地址。不需要参数！
 */
function getLocation() {
  var phoneNumber = prompt('What is your phone number?');
  var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
  var phoneMatches = phoneNumberPattern.exec(phoneNumber);
  var areaCodes, areaCode, locationName;
  if (phoneMatches) {
    areaCode = phoneMatches[1];
    areaCodes = getAreaCodes();
    locationName = areaCodes[areaCode];
  }
  return locationName ? locationName : 'somewhere';
}
```

清单 10.7　重构后的 `getLocation`，没有更新注释

```
/**
 * getLocation 请求用户输入一个电话号码，
 * 然后基于电话号码返回一个地址。不需要参数！
```

```
 */
function getLocation(phoneNumber) {
  var phoneNumberPattern = /(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/;
  var phoneMatches = phoneNumberPattern.exec(phoneNumber);
  var areaCodes, areaCode, locationName;
  if (phoneMatches) {
    areaCode = phoneMatches[1];
    areaCodes = getAreaCodes();
    locationName = areaCodes[areaCode];
  }
  return locationName ? locationName : 'somewhere';
}
```

用文档来找 bug

记录意图的一个好处是，你的文档可以帮你在代码中找 bug。如果你正确的将意图记入文档，而你的代码并不匹配你的文档，有时候可能是你的代码有问题。例如，清单 10.8 中的文档是说我们要得到总价，暗示着我们会用 `itemsQuantity` 乘上 `itemPrice`。然而，代码并没有做乘法，而是将 `itemPrice` 和 `itemsQuantity` 相加。结果不应该是 13，而应该是 42。文档帮我们发现了一个 bug。

清单 10.8 正确记录意图可以帮助发现代码中的 bug

```
var itemPrice = 7;
var itemsQuantity = 6;

// 获取总价
var total = itemPrice + itemsQuantity;
console.log('The total price is ' + total);
```

为自己写文档

你可能认为只有当你和其他人一起做项目时文档才有用。如果你独自工作，你已经知道你可能要写进文档里的所有内容，写文档还有什么意义，对吧？事实上，写好文档还是有很多好处的，即使你一个人在项目上工作。

你的记忆力有多好

假设你没有完美的记忆（主要是因为我觉得只有电视剧中的人才可能拥有完美

记忆）。如果我错了，你从不忘记任何事情，你可以跳过本节，直接看下一节。我们其他人都会忘记。有时候忘事没什么大不了的。为什么 6 个月前的一个周二你选择穿绿色的 T 恤？可能因为那天是圣帕特里克节。可能那是唯一一件干净的 T 恤，可以用来配你的鞋。可能那就是唯一干净的 T 恤，没了。为什么 6 个月前你穿了绿色衣服很可能没什么大不了。然而，编程的时候，忘记为什么做了某个决定可能会是个大问题。为什么当时选择将数据表示成数组的数组，而不是其他数据结构，比如对象的数组呢？是因为你没意识到对象数组也是一个选项吗？是因为你认为数组的数组会让代码运行更快？是因为你用的某个开源项目，接收的数据结构就是数组的数组？

你所做的决定，以及做决定的原因，在你做这个决定的时刻，可能会很明显。但是，事情会随时间变化。6 个月以后，你可能学到了新的表示数据的方式；你可能意识到如果选择更简单的方式，代码会运行的一样快；你可能开始用不同的开源项目，接收不同的参数。6 个月过去了，你面临一个新的决定，你会记得第一次决定背后的原因吗？我猜答案应该是“不会”（如果你有完美记忆，你应该已经跳过本节了）。人是物非，你当初的决定还合理吗？如果你不写文档，就要重新做一遍所有的工作和调研，深入研究这个决定，以确保正确。

为了学习而记录文档

写好文档要求你真正理解你的代码在做什么。如果你不理解自己的代码，你怎么可能将它的用途写成文档呢？因此，编写文档会迫使你对自己的代码如何工作相当了解。在各种场合，我在写代码文档时会学到某些事情，让我意识到，我把一些事做错了。我发现了问题，同时也修复了代码。这是个双赢的结果。

超越注释的文档

注释通常是记录一段代码如何工作的最佳地点。注释在你要记录所有代码如何一起工作时就不那么有用了。流程图、图表、手绘，以及记叙文都要更适合这类文档。第 5 章“数据（类型）、数据（结构）、数据（库）”中的图 5.12，记录了图书馆数据库的结构，而第 7 章“何时使用 `If`、`For`、`While`”中的图 7.2 记录了 `prompt.js` 如何工作。像这样的图表，用来描述程序的长远愿景和想法时很有用。而注释对于琐碎的细节更有用。

现在是时候给我们的 kittenbook 文档升级了。首先，我们要添加项目描述到 README 文件。一般 README 文件是从开发者的角度描述软件的。README 文件包含程序的意图和目的信息，也可能包含程序的设计信息。你可能注意到了，在第 4 章“构建工具”中，每次运行 npm install，npm 会提示说你没提供 README.md 文件（见图 10.2）。现在就添加一个，写上任何你觉得有价值的东西。清单 10.9 展示了 README.md 文件可能看起来是个什么样子。

清单 10.9 kittenbook 的 README.md 示例

```
#kittenbook
Chrome 扩展程序，使用下面的图片替换新鲜事物的图片，可以让你的 Facebook 时间线更加可爱：

- 小猫
- 小狗
```

注

Markdown

README.md 是一个 Markdown 文件。Markdown 是一种简化的编辑方式，用来编写 HTML，Markdown 使用符号而不是标签。不过，你不必使用任何符号：你可以用纯文本来写 README.md。如果你要尝试一些 Markdown 的功能，看下 http://daringfireball.net/project/markdown/，并使用 http://daringfireball.net/projects/markdown/dingus 查看从 Markdown 转换过来的 HTML。图 10.3 展示了清单 10.9 中的 Markdown 被渲染成的 HTML。

图 10.2　NPM 在没有 `README.md` 文件时以提示的方式尝试提醒你写代码文档

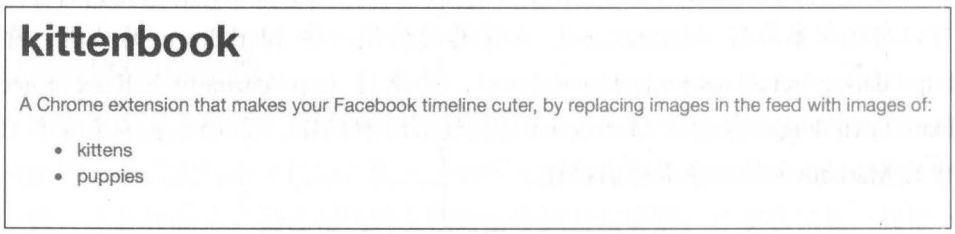

图 10.3　清单 10.9 中的 Markdown 渲染成 HTML

改进 kittenbook 文档的下一步是为每个函数写描述。如果你使用正确的格式写描述，就可以用 Grunt 来生成看起来很不错的文档（使用名为 JSDoc 的程序）。清单 10.10 展示了这个格式，使用 `prompt.js` 中的 `validatePhoneNumber` 函数。

清单 10.10　使用 JSDoc 给 `validatePhoneNumber` 生成文档

```
/**
 * 检测电话号码的有效性
 * @method
 * @param {string} phoneNumber 要检测的电话号码
 * @return {boolean}
```

```
 */
function validatePhoneNumber(phoneNumber) {
  return phoneNumber.match(/(?:1-)?\(?(\d{3})[\-\)]\d{3}-\d{4}/);
}
```

文档就直接放在它们要描述的函数上面。第一行是函数功能的概述。其他行对于JSDoc有特殊意义。@method表示你要表示一个方法或者函数。下一行(以@param开头)描述了函数的参数：{string}表示参数应该是一个字符串，phoneNumber是参数名，最后一部分是参数的描述。最后一行（以@return开始）描述了函数会返回什么，在这里，是一个布尔值。

在你给一些函数添加了一些JSDoc风格的文档后，你需要安装grunt-jsdoc（见清单10.11），并添加一个jsdoc任务到Gruntfile.js（见清单10.12）中。当这些都做完，你可以用命令行在kittenbook目录下运行grunt jsdoc，然后运行open doc/index.html（或者双击资源管理器中的index.html文件图标）打开grunt-jsdoc生成的网页（见图10.4）。当你给所有函数添加好文档，你就可以在一个（很容易找到的）地方看到所有文档（见图10.4）。你的文档马上就看起来既整齐又专业。你要做的就是遵守清单10.10中的模式，给项目中的其他函数填上文档。

清单10.11 安装grunt-jsdoc

```
kittenbook $ npm install grunt-jsdoc --save-dev
```

清单10.12 向Gruntfile.js中添加jsdoc任务

```
module.exports = function(grunt) {
  grunt.initConfig({

    ...

    jsdoc: {
      dist: {
        src: ['js/*.js'],
        dest: 'doc'
      }
    }
  });

  // 加载 Grunt 插件

  ...
```

```
  grunt.loadNpmTasks('grunt-jsdoc');

  // 注册任务
  grunt.registerTask('default', ['concat', 'jshint', 'copy']);
};
```

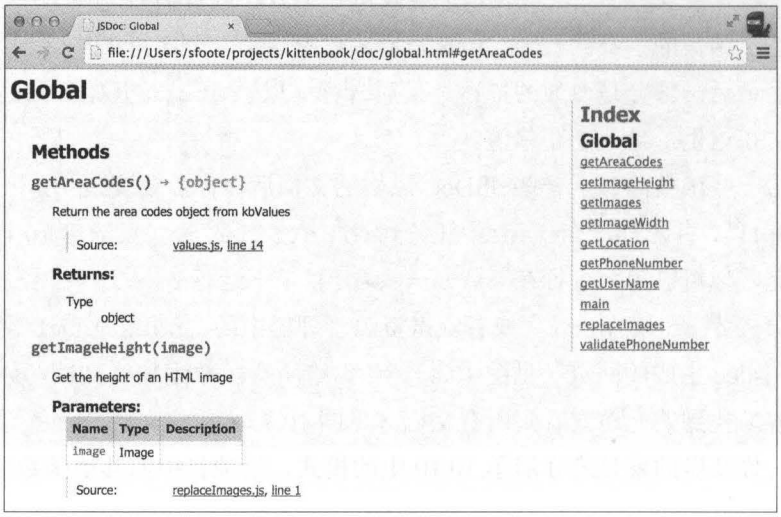

图 10.4　JSDoc 将文档变成漂亮的网页

给别人写的文档

如果你和其他人一起工作，你还需要满足他们对文档的需求。在大脑中想象一个听众，这样写下的文档要更加直接和有用。问问自己，“当一个新的开发者读着这注释时，他能理解我为什么做这个决定吗？”如果你还没和其他人一起工作，你仍要为别人写文档，最终，可能会有人接手你的项目。我搭建的第一个大项目（就是那个我一直在说的网站），就是我一个人做了大概两年。我几乎没花时间写文档，一部分原因是我一直忙于写代码，另一部分原因是我没有意识到写文档的价值；我是唯一一个在项目上工作的人。这个网站是我在学校工作的一个大作业。在我毕业前的 4 个月，我意识到，如果我不采取行动，在我离开后，这个网站就会死掉。我花了最后 4 个月的时间补上了之前两年代码的文档。我这辈子也没办法想起来当时我为什么要写那些代码。我不得不花大量时间重新学习我的代码在干什么，这样我才

能给它们写文档。写 4 个月的文档绝对不是什么好玩的事。我真希望当时在我写代码的时候就写好了文档，为了我自己，也为了我的继任者。

记录你的决定

把决定写成文档对于他人比对你自己更重要。文档证明你做了个明智的决定，并且考虑了其他方案。这样，当其他人需要修改你的代码时，那个人就会知道为什么你做了这样的决定，以及能否安全切换到其他方案。清单 10.13 展示了一个具有良好文档的决定。

清单 10.13 使用 `while` 而不是 `for`，文档记录原因

```
var letters = ['t', 'p', 'i', 'c', 's', 'a', 'v', 'a', 'j'];
var currentLetter;
// 使用 While 循环，而不是 for 循环，
// 因为数组在循环中被修改，在 for 循环中修改数组会破坏整个逻辑
while (letters.length) {
  currentLetter = letters.pop();
  console.log(currentLetter);

  // 向数组添加新的字母
  if (currentLetter === 'c') {
    letters.push('r');
  }
}
```

一天晚上，在我下班回家后，我收到一条短信，说我需要立即修复一个阻塞 bug（阻塞就是一个很不好的 bug，以至于你要停下所有其他正在做的事情来修复它）。这个 bug 是由于一行不是我写的代码；我甚至完全不了解那段代码。删掉那一行代码看起来修复了这个 bug，看起来也没有对程序的其他地方有什么影响。这让我感到奇怪，为什么那一行代码一开始被加到代码中。一定有什么原因，我不敢移除那一行，就是以防删掉它会引起其他 bug。加入那一行有详细文档记录，我就不会有这个问题了。

记录你的资源

文档不只用来解释代码在做什么，以及做决定的原因，还可以是关于你的想法。如果你在努力解决一个问题，并且在论坛或者博客中找到了一个解决方案，你应该在注释中提及这一灵感来源。这么做也算是吃水不忘挖井人；归根结底，你是在用其他人的解决方案。如果你的灵感来自于一个可靠来源，这样的声明还可以给你的

代码带来可靠性（就像是你在根据沃伦·巴菲特的建议做投资决策）。你还让其他人也了解到这个资源，读到这段注释的开发者也可以访问这个资源。这些开发者可以读到原始的博客，也能从中学习。

为了教学而写文档

通过阅读别人的代码，特别是注释，我学到了很多编程知识。你在用到一个好玩的技术时，就有机会向跟随你的开发者们传授这项技术，并展示为什么它很棒。花些时间，写下教学文档，你自己也会学到很多相关的知识。清单 10.14 展示了我最喜欢的技术之一，遍历一个数组。我是从 JSHint 文档上学到这个技巧的。

清单 10.14　在 `for` 循环的条件中赋值

```
var letters = ['a', 'b', 'c'];
var letter, i;

// 没有使用数组的长度，而是在 for 循环的条件部分进行赋值，当没有元素要被赋值时，条件就会变为假，这样循环就会
// 停止。作为额外奖励，你不需要在 for 循环中执行赋值
for (i=0; (letter = letters[i]); i++) {
  console.log(letter);
}
```

总结

本章讲了很多（还有更多！）你曾经没想过要学的内容，教你如何给软件写文档。文档可能不是编程最激动人心的部分，但它确实很有价值，值得好好写。本章你学到了：

- 在文档中记录意图
- 为自己记录文档
- 超越注释的文档
- 为他人记录文档

下一章中，你会学到：

- 规格说明
- 迭代计划
- 软件架构

11
计划

注

项目：开始计划你的下一个项目。将计划发布到网络上，听听大家的反馈。

我之前说过，编写代码只是编程的一部分，甚至不是最难的部分。你可能会觉得奇怪：如果写代码不是编程最难的部分，那什么才是？答案可能是做计划。在写代码时，你是对程序的一部分如何工作做出细小的决定（例如，那些指令应该包含进 `if` 代码块）。在做计划时，你会对程序整体如何运行做出较大的决定（用什么编程语言？用什么数据结构？如何存储数据？）。

做计划解答了一个重要问题：程序要解决什么问题？如果你无法回答这个问题，你就没有完成计划。做计划还要解答另一个问题：这个程序不会解决什么问题？正如第 8 章“函数和方法”中的函数一样，程序应该有一个明确的目的，不要尝试解决目的之外的问题。软件程序是一系列模块一起工作，达成同一个目标。一旦你识别出这个目标，就可以对系统有更清晰的认识，能够明确地识别出什么是实现目标所必需的。将系统作为一个整体来考虑，可以帮你定位到程序中可能给你带来麻烦的部分。这个过程会让你得到一个真实的估计，构建项目大概需要多久。

三思而后行

我希望你已经体会到编程的乐趣所在。写出可以工作的代码着实让人兴奋。当我尝试解决一个困难问题，最终让我的代码成功工作之后，我常常会将双手举向空中，就好像我刚刚在一场篮球比赛中投中制胜一球似的（那种宅男景象你应该可以想象）。写代码真的特别有趣，特别有成就感，以至于一旦你启动一个项目，你就会迫不及待想要开始写代码。项目的计划和设计阶段很容易被忽略。

设计阶段特别容易被跳过，是因为设计软件很困难，让人很困惑，很抓狂。你必须用你的想象力，必须想象你的软件在完成后看起来是什么样子。然而，设计软件并不意味着你要在文本编辑器里搭建软件之前，就在大脑中搭建整个应用。当你设计一辆车时，你不需要考虑用什么样的螺丝钉来固定引擎；你可以在真正实现设计的时候再想办法解决细节问题。然而你在设计时，你确实需要决定引擎被固定在哪里，传动装置如何与汽车的其他系统连接在一起。在构建之前从整体来设计系统，会帮你确保不会把备胎放在引擎应该放的地方。好的设计会带来更干净的代码。所以，尽管写可工作的代码很有趣，很激动人心，代码通常会在良好设计的系统中工作得更好。

创建规格说明

设计软件的一种方式是先写一个详细规格说明。规格说明是一个用来解释你计划构建什么的说明。在写规格说明时，要在不陷入细节的情况下，尽可能明确。规格说明应该包括像如何处理错误这样的细节。例如，kittenbook 的规格说明会包含一些像“如果用户输入一个不合法的电话号码时，要给用户第二次机会。如果电话号码还是不合法，就接受它，并将用户的位置设置成“某地”。

规格说明可以非常详细，也可以仅仅是对程序的顶层描述。不管是哪一种情况，写下什么东西，都会帮助你控制项目的适用范围。你应该实现所有规格说明中的功能，而规格说明以外的功能都不要做。要意识到，在写需求规格说明时，每一个你漏掉的决定，在你写代码的时候还是要补上。在某些情况下，这样挺好，因为在你已经搭建了一部分程序之后，再做一些决定时，会处于更有利的位置。规格说明不要对每一个决定都面面俱到，但应该要足够详细，让你在写代码时能够起到足够的指导作用。随着时间和经验的积累，你会找到其中的平衡。

设计架构

规格说明书告诉了你要搭建什么内容，但没告诉你如何搭建。规格说明书也不会帮你把所有细节组装在一起。事实上，规格说明很少会包含程序各个组成部分的细节信息。在创建规格说明之后，开始写代码之前，就是为你的程序设计架构的时候。

在设计软件架构时，你要识别出程序的全部组成部分，然后决定这些不同的部分该如何组装在一起。架构决策在我看来，是计划阶段最难的部分。你必须动用你的想象力，预见软件在完工时应该是什么样子。你要识别出想象中的软件的各个部分，然后找出组装它们的最佳方式。尽管可以在开始写代码之后再做这些架构决策，但每次我尝试这么做，我都会遇到这样两个问题：

1. 某些关键组件在逻辑上无法放到程序内（见图 11.1）。要么我得重写大量已经完成的代码，将不适配的组件包含进来；要么我就得将这个组件放到它可以放的地方，然后做大量的额外工作来让它正常工作。

2. 在解决设计问题时，我们的思考方式和解决编码问题时的思考方式非常不同。解决设计问题更像是一个创造性的，甚至是艺术性的过程，而解决代码问题是分析性的、具体的。如果你没有在开始写代码之前解决掉架构设计问题，你最后就不得不尝试同时解决代码问题和设计问题，结果两者都没办法解决好。

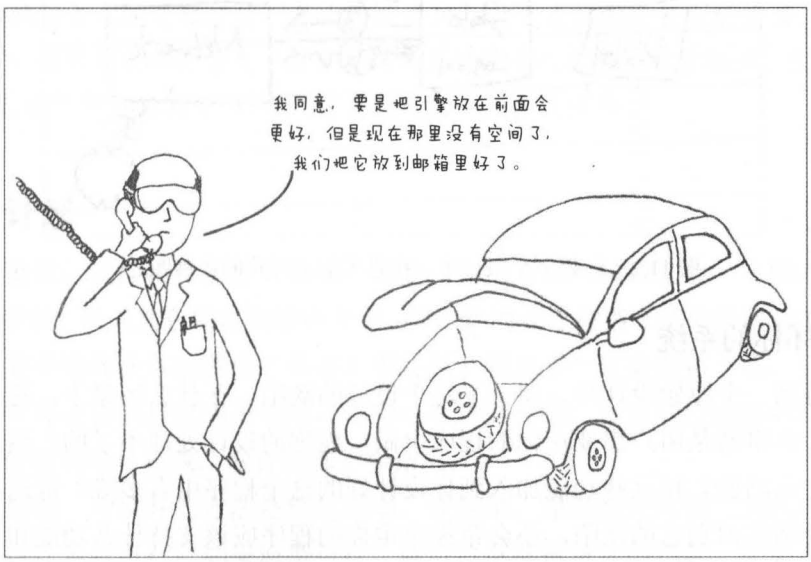

图 11.1 如果你在设计前就开始搭建，最后可能会把东西放到原本不属于他们的地方

画示意图

如果你在设计软件时遇到困难，就把设计画出来。画图可以帮助你将问题可视化，这样你就可以看到各个组成部分，以及它们之间如何关联。我喜欢画图（我在

这本书里的图可以证明），尽管我画得并不怎么好（还是可以用本书中的图来证明），所以说画图对我来说确实很有用。画图最重要的一点是快。你不需要搭建或者写任何东西出来。可能画第一幅图时没有设计得很好，但至少你没有浪费时间；拿出另一张纸，重来就好了。当你得到了好的设计，你画的图就可以作为文档使用。图 11.2 是我在最近规划的一个项目中所画的各种架构图之一。

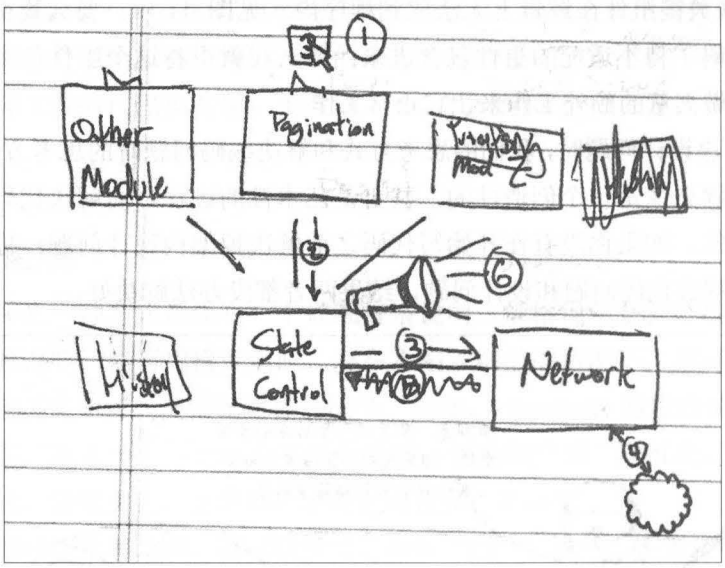

图 11.2　最近一个项目中，在思考架构时画的各种图之一

尝试破坏你的系统

在得到一个原始设计时，想一下这个设计的缺陷。在什么场景下，这个系统会崩溃？什么事情是用户想要在你的程序中做，而你的设计处理不了的？用户在未来会想要什么功能？将这些功能加入到你设计好的这个程序中有多难？像这样思考你的设计会帮你识别它的缺陷，还会帮你决定你的程序应该支持什么功能和用例。你不用支持每一个功能或用例，但通过这个过程，你应该对支持哪些功能有个比较准确的判断。

在搭建 Web 应用时，很多时间都会花在支持旧的浏览器，特别是 IE 上。如果设计没有考虑到 IE，你的应用可能对于 IE 用户是不可用的。如果你想要在设计完成之后再考虑加入 IE 的支持，就会遇到很多麻烦。支持 IE 的解决方案应该在设计阶段就

考虑进来。另一方面，你可能决定你的应用不再支持 IE 用户，那么就在设计中显式的排除 IE。不管哪种情况，你都考虑了程序可能的崩溃情况，并且在设计中体现了这些情况。kittenbook 的设计不需要考虑 IE，因为 kittenbook 是 Chrome 扩展，它不可能用在 IE 中。

> **意大利面式的代码**
>
> 我喜欢吃意大利面。甚至今天中午就吃了意大利面（不开玩笑）。意大利面非常好吃。但代码中的意大利面就不那么好了。意大利面式的代码是指代码就像一碗意大利面一样，一团乱，绕来绕去，互相纠缠。你不可能对单个的一根面条做点什么而不影响整碗面。到第 8 章为止，kittenbook 代码都很像意大利面式代码。意大利面式代码的问题在你想要修复 bug 或者添加新功能时就会暴露出来。当你要添加或者修改代码的某个地方，你会发现要达到目标，你不得不在代码的所有地方都做修改才行。我的第一个 Web 应用就是意大利面式代码中最差的那种，我原以为编程本来就是那样的。每次修改应用，对我来说都是巨大的痛苦，因为我并没有意识到有更好的方式。有了好的软件设计（以及面向对象编程，在第 15 章“高级主题”中会学到），你可以完全避免意大利面式代码。

迭代式计划

对我来说，软件设计的最大挑战，就是要在搭建任何东西之前，在脑海中尝试设计出完整的、对未来有保证的解决方案。假如我的设计在某个部分是错的怎么办？假如未来并不像我计划的那样怎么办？事实是，我几乎一定会做错一些，但其实这是好事。在你意识到自己的失误时，你应该继续做计划。做计划应该在你写代码之前就做，但直到项目完成，做计划都不会结束。

做计划的一种很好的方法是创建原型。在这里，原型是快速编写的代码，用来证明你的设计是否可行。如果原型可行，就可以继续推进你的设计。如果不可行，你可以回到画图板做设计，避免投入太多时间去实现有缺陷的设计。原型也可以让设计软件变得更有意思。

为扩展设计

无法预见未来的另一个问题在于你没办法预料到每一个你或你的用户想要添加到程序中的功能。你应该以容易添加（或移除）新功能的方式来搭建程序。一个可扩展程序就像是带有 HDMI 接口的电视，可能电视并没有内置 DVD 播放器，但电视可以扩展，这样你就可以接入一个 DVD 播放器。当用蓝光播放器替换 DVD 播放器时，你不需要买一个新电视，因为你只要移除 DVD 播放器功能（拔下插头），然后添加蓝光功能就可以了。类似的，如果你要添加 Roku（一个流媒体播放设备），你不需要重新搭建整个电视，只要插入 Roku 就可以了。

你的优先级是什么

人不可能得到所有。这句话对于软件来说和生活中其他情况下一样正确。事实是，你必须在用户体验、性能、安全、可伸缩性及截止日期之间做出妥协，不过这一事实在设计结束、开始开发之前并不那么明显。然而，考虑这些事情应该是软件设计的一部分。你为软件选择的架构很大程度上依赖于这些选项中哪一项更重要，而排期依赖于你将要并能够做出的妥协。

用户体验

如果你在搭建软件，有人即将使用这款软件（这个人很可能不是你），这就意味着你有用户。如果你有用户，你就应该为他们建立良好的用户体验。建立良好的用户体验需要花费时间，而用户体验一定是计划和优先选项的一部分。在设计软件时，你应该从下面几个方面考虑用户体验：

- 你的用户是谁？他们是程序员吗？他们是相对有经验的计算机使用者吗？他们是计算机新手用户吗？对于每一类用户，你都需要设计不同的软件。
- 软件符合直觉吗？一般用户是否能够理解程序如何工作，尤其是在没有阅读很长的说明之前？
- 应用是否尊重用户隐私？隐私和安全关系紧密。如果你向用户请求数据，你需要确保数据安全，如果用户期望数据完全保密，你需要设计你的软件，保证数据的传输和存储是安全并且秘密的进行。
- 是否可用？软件不用解决所有问题。你不需要搭建出你或者你的用户梦寐以

求的软件。但你搭建好的功能应该随时可用。

性能

当谈到软件的性能时，是指软件执行任务的速度有多快。有几种方式可以提升性能，他们依赖于你工作的环境。不过，通常来说，你可以通过减少程序执行任务必须运行的指令数量，来提升性能。例如，如果你的程序用到了循环，将循环中所有不必要执行的指令都移除。学习算法的一个重要部分，就是理解为什么程序运行得慢，以及如何让它们运行得快。如果你对学习性能相关的知识感兴趣，你可以找到很多在线算法课程（见第 13 章“授人以渔：如何用一生学习编程”中可以找这些课程的更多信息）。

为了性能而设计软件的最好例子就是谷歌搜索。从前到后，谷歌搜索的体验都是为了速度而设计。首先，位于 https://www.google.com/的网页加载得非常快，因为上面几乎没什么东西，基本上，就是谷歌的 Logo、搜索框，以及两个按钮。这种极简主义设计的原因之一是这样确实快。当你执行搜索时，搜索结果几乎瞬间就返回，因为谷歌写的搜索和结果排序代码，是为了速度而设计的。性能对于谷歌软件来说是最高优先级。

安全

如果你有用户，安全性就需要优先考虑，特别是你向用户要数据。不幸的是，另外有些人用你的软件是为了利用你的用户（这些人就是恶意的黑客）。你有责任为你的用户保障他们的数据安全，并且阻止黑客使用你的软件做坏事。这也是为什么微软、苹果，以及 Adobe（以及其他很多软件公司）为他们的软件定期发布“安全更新”的原因。加入安全性是一个优先选项，那就应该成为软件设计的一部分。例如，你应该把软件设计成用户密码非明文存储，密码应该加密。

伸缩性

软件设计的一个主要关注点是伸缩性。如果你只有 15 个用户（或者 150 个），或者你的软件只会用在执行一些小任务上，你不太需要在设计软件时担心伸缩性。如果相比软件的容量，你的用户群增长速度出乎意料的快，你就很可能需要重新设计并且重写软件，但这是个幸福的烦恼。研究和设计高可伸缩性系统远远超出了本

书的范围，你只要记住，伸缩性是一项重要的设计关注点。

截止日期

你的软件需要多久完成？有时程序的截止日期具有最高优先级，那么软件的设计就要基于这个截止日期作出修改。你可能不得不牺牲用户体验、性能、或者伸缩性，来适应截止日期的要求。因此，将截止日期设为最高优先级通常会导致低质量的软件。软件的全部优先选项都应该被仔细考虑，然后你可以基于那些优先选项设定一个现实的截止日期。这可能是一个迭代过程：当你设置的截止日期看起来太遥远，你可以移除一些性能或者用户体验功能，然后重新估算时间。截止日期在设计和做计划时可能有些帮助，但通常不应该是最高优先级。

平衡的艺术

这些提到的、没提到的优先选项相互竞争，再说一次，你无法得到全部。不过，有时候相互竞争的优先选项可以协调一致。例如，提升性能可能提升用户体验。但如果你过于强调提升性能（比如，通过移除某些重要功能），用户体验就会受损。有些时候，你需要决定这些优先选项的相对重要程度，这个决定最终决定了软件的设计和功能。

识别并创建限制条件

在你设计软件时，你需要识别系统的局限性和限制条件，并且你应该设置自己的限制条件。在第 9 章“编程标准”中你感觉到了设定标准的重要性。设定标准就是在设定限制条件，也就是说，不是什么都可以，即使一个东西是可行的，但不意味着就是个好主意。举个例子，kittenbook 的第一个版本完全清除了 Facebook 上每个页面的所有东西，这是个多么糟糕的体验。对于教学，这很有帮助，而且某种意义上说还挺酷的，但它确实是个很不好的用户体验。避免这类不好体验的一个比较好的自我约束条件是“绝不打搅用户”。这样，如果未来你有个想法，想在发生错误时显示一个闪动的红色消息，别忘了你的自我约束“绝不打搅用户”。

知道可以做什么，不可以做什么

理解你所在环境的能力，是非常必要的。你也许可以在软件中做很多酷炫的事

情，但你可能并没有意识到这种可能性。你知道你可以在谷歌浏览器中创建桌面通知（如图 11.3）吗？有了桌面通知，你可以很容易通知用户，应用中正在发生什么事情，即使用户在当前没有打开窗口。假设你在搭建一个 To-Do 列表应用，To-Do 列表中的每一项都有截止日期。你可以创建桌面通知，提醒用户截止日期快到了，但仅当你知道桌面通知这个功能存在才能做。如果你理解了系统能做什么，你就可以在软件的设计中包含这些能力。

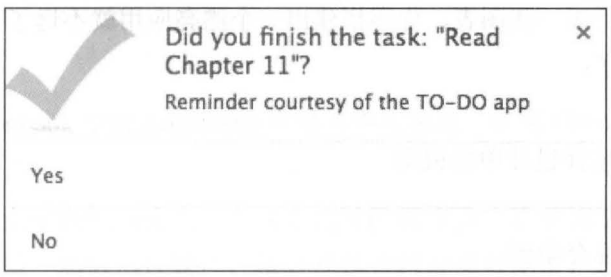

图 11.3 了解你的环境的能力

计算机可以做很多酷炫的事情，但它们并不是万能的。理解系统的局限性和理解它们的能力一样重要。移动设备可以通过摄像头、地理位置，以及设备方向做很多很棒的事情，但移动设备的内存和电量相对小很多。当你给移动设备设计应用时，内存和电池的使用就是重要的限制条件。当你意识到你设计的软件，因为死板的系统限制导致技术上不可行时，你一定会感到很伤心。你应该在这些限制条件之内设计软件，而不是对抗这些限制条件。

权限

Chrome 扩展、安卓应用，以及 iOS 引用都有权限的概念。通过权限，系统环境设置了限制条件，你必须显式的请求权限来移除这些限制条件。举个例子，在 Chrome 扩展的环境中，权限限制会阻止你在特定网页加载 JavaScript。在 kittenbook 的 `manifest.json` 中，我们向用户请求在`*://www.facebook.com/*`的网站上加载 JavaScript 的权限。用户在安装这个扩展时，必须同意我们请求的权限才行。权限可以帮助你给应用设置一个合理的限制条件，只有当你确实为了实现应用的功能而打破限制时，才想办法去掉限制。权限还可以帮助你的用户更好理解应用的能力和限制。

总结

本章你学到了很多关于如何做计划和设计软件的知识。真正消化这些概念的最佳方式就是实践它们。如果你想，你可以从头开始重新设计 kittenbook，但我觉得你做自己的项目会更加有趣。在你想到一个好项目的时候，识别出它的不同组成部分，然后为如何将这些部分组合在一起制定一个计划。优先级是什么？你要在什么环境中构建应用？（Web 应用、Chrome 扩展还是 iPhone 应用？）有什么限制条件？把这些都写下来，然后画一些图表，你离搭建出一个漂亮应用就不远了。

本章你学到了：

- 软件架构
- 优先级和软件设计中的权衡
- 限制条件

下一章中，你会学到：

- 手工测试软件的策略
- 如何写代码来测试你的代码
- 在出错时如何使用调试工具

12

测试和调试

注

项目： 为 kittenbook 中的 JavaScript 代码写单元测试。使用 Grunt 运行单元测试。

本章中，你会学到如何成为一个真正高效的程序员。软件测试是一种实践，首先设定软件的期望，然后运行软件，看软件是否符合那些期望。这可能听上去像是一个专横的家长，但软件测试其实是对软件有好处的。当期望没有达成，你可以修改软件，直到它正确工作。当全部期望都满足，你就可以准备好把软件发送给全世界了（嗖嗖嗖——它们成长得好快）。

要修复不符合预期的软件，你必须找到为什么软件发生故障。探索这一奥秘的过程叫作调试。使用正确的工具，调试其实可能很有趣。你在寻找的 bug 就像是一个老练的贼，而你就是侦探。如果你明智地使用工具，就可以沿着线索，找到凶手。好的测试和好的调试可以让你的程序对你和你的用户都更好。

手工测试

手工测试的想法相当直接。写一写代码，然后运行程序。可以工作么？它做的是否是我期望它做的？看起来很简单，不过手工测试其实很强大。手工测试过程中，你其实是在使用你自己写的软件。如果你要打一把椅子，难道你不要试着坐在上面，以确保它很舒服，而且可以支撑你的重量么（见图 12.1）？还存在其他类型的测试，但都没有办法可以替代真正使用你的软件。

图 12.1　椅子坐起来并没有看起来那么好

边做边测

要经常运行你的程序。我喜欢每次对代码作了修改就运行一遍程序。每次你运行程序，都要检查一下程序做了你期望它做的事。如果你在两次运行之间做了一大堆修改，你会很难记住检查每一个修改，看是否正常工作。而且如果其中一个修改影响了程序正常运行，你就难以找出哪些变化是有问题的。我就遇到过这种问题，一次性写了很多代码，最后写完之后才测试。我有时会写一堆代码不测试，并且想着，“我知道我在做什么，这些应该都没问题。”只有很少的情况下（过去 7 年中的三次），它确实能工作。然而通常，我都会在追踪 bug 的过程中，挣扎着撤销几乎所有做的修改。通过经常测试得到的小胜利，要比写完所有代码才测试而得到的大胜利（极少数情况下）要好得多。

尝试些疯狂的事

在你通过手工测试查找 bug 时，你会找到，然后在代码中修复这个问题，然后再次运行程序验证为了修复 bug 做的修改。有时你会做一个修改，运行程序，发现没什么变化，尽管你的修改应该修复了这个 bug。然后你就回到代码中，做些别的修

改，然后再次运行程序。修改看起来还是没什么效果。这种情况我遇到过几次，可能会让人感觉抓狂。我建议做些疯狂的事。删掉整个文件（在备份之后），或者写一写你明知道会出错的代码。然后在此运行程序。如果你最后的疯狂修改看起来没什么效果，你就知道，程序并没有真正在读取你修改的文件。图 4.7 展示了忘记在执行之前编译代码，会导致这个问题，当然还有其他可能的解释。例如，我有一次把我的整个项目代码复制到计算机的另一个目录，下次我想要在那个项目上工作时，我忘了我已经将目录移动到了新的地方，我就开始编辑旧的项目目录里的文件。在你开始拿头撞墙之前，尝试些疯狂的事情，看看你的程序是不是真的在读取你正在做的修改。

吃你自己的狗粮

在软件公司的雇员通常用他们公司生产的软件来工作。这种实践被冠以一个影响食欲的名字，吃狗粮，或者“吃自己的狗粮”。这个故事来源于 Kal Kan 宠物食品公司的 CEO，在每年的股东大会上会吃一罐他们自己的狗粮，由此，使用自己的产品就有了这个名字。我不推荐吃狗粮，但我确实推荐使用你自己搭建的软件。如果你知道你必须依赖你的软件，搭建高质量产品的动力就会提高。

在我做会计的时候，很讨厌记录我的计费小时数，但我必须精确地记录计费小时，这样才能向客户收取我所做工作的费用。我决定搭建一个 Chrome 扩展，通过提醒帮我记录小时数。在我搭建这个扩展的时候，我还得同时追踪我的小时数，所以我还是用笔和纸来记录，因为我知道它们可用，尽管用得很痛苦。终于，我觉得我的扩展已经准备就绪。第一周，我用扩展和纸笔同时记录我的小时数，就为了确保软件不会漏掉什么内容。那一周，我发现并修复了很多 bug，在之后一周一开始，我觉得有足够信心可以扔掉纸笔装备，完全依赖我的扩展来记录时间。两周之后，又出现并修复了很多 bug，我就准备好把这个扩展分享给我的同事们了。我花在“吃狗粮”的那些时间，对于这个扩展的质量是非常必要的。

自动化测试

尽管手工测试目前是，而且未来也还是确保程序工作的必要部分，自动化测试还是更有趣一些。你现在是程序员了，你可以把所有事情自动化，从复制文件到铺

床，为什么不把测试也自动化了呢。我们要写测试我们代码的代码。有点懵，是吧？我想也是，但给你几个例子，你就会发现事情没有那么糟。测试代码其实就是运行全部或者部分应用代码，并比较实际结果和期望结果。测试整个应用一起工作的叫作集成测试，而测试应用的某个部分在某个隔离环境下的工作情况，叫作单元测试。

单元测试

我觉得单元测试这个名字有点让人困惑。什么是单元，我该怎么测它？事实上，我完全没有搞清楚，以至于我在很长一段时间内都决定不写单元测试。尽管，现在我已经学了很多单元测试的东西，而且我觉得这个名字很准确了，然而它还是很让人困惑。从我们的目标来看，我们可以把单元简单的看作是一个函数。所以要测试一个函数，我们需要运行这个函数，给定一些输入，然后看看函数是不是按照期望的行为工作。

假设我们有个函数叫作 sum，我们要给它写单元测试。在我们开始写单元测试之前，需要确保我们清楚地知道 sum 要做什么（看规格说明可能有所帮助）。对于这个例子，我们假定 sum 接受两个数字作为输入，然后返回两个数字的和。我们不想测试所有两个数字的可能组合，这样的努力注定是徒劳的。既然测试每一种可能的输入是不可能的，你应该从每一类输入中，挑一个作为样例进行测试。我觉得 sum 函数有以下几类输入：

- 两个正数
- 两个负数
- 一个正数和一个负数
- 0 和 0
- 0 和一个正数
- 0 和一个负数

尽管还可能有其他类型的输入（比如很小的正数和很大的正数），这个列表已经足够全面了。现在我们来写一些测试，从清单 12.1 的例子开始。

清单 12.1 如何用代码测试代码的例子

```
// 真正sum的定义
function function sum(x, y) {
  return x * y;
```

```
}

// 我们会把所有的测试放到 testSum 函数中
function testSum() {

  // 用来跑测试的工具函数
  function test(x, y, expected) {

    // 实际运行 sum 函数，并保存结果
    var result = sum(x, y);

    // 结果是我们期望的吗?
    if (result === expected) {

      // 是的，测试通过
      console.log('Pass!\n');
    } else {
      // 不是，测试失败，然后会输出一些信息
      // 关于到底发生了什么，以及我们期望发生什么
      // console.error 和 console.log 的工作原理差不多，但它意味着打印出错日志

      console.error('FAIL: expected the sum of ' + x + ' and ' + y +
                    ' to be ' + expected + ', not ' + result + '\n');
    }
  }

  console.log('Testing sum of two positive numbers');
  // 期望 2 + 2 等于 4
  test(2, 2, 4);

  console.log('Testing sum of two negaive numbers');
  // 期望 3 + -2 等于 5
  test(-3, -2, -5);

  console.log('Testing sum of one positive and one negative number');
  // 期望 3 + -5 等于-2
  test(3, -5, -2);

  console.log('Testing sum of 0 and 0');
  // 期望 0 + 0 等于 0
  test(0, 0, 0);

  console.log('Testing sum of 0 and a positive number');
  // 期望 0 + 3 等于 3
  test(0, 3, 3);

  console.log('Testing sum of 0 and a negative number');
```

```
  //期望 0 + -3 等于-3
  test(0, -3, -3);
}

// 最后，调用 testSum 执行所有测试
testSum();
```

清单 12.1 里面有很多要详细看的内容。首先，尝试运行这段代码，这里，我建议把代码保存成名为 testSum.js 的文件，然后用 Node 从命令行执行这个文件。导航到含有 testSum.js 文件的路径，然后运行 node testSum.js，代码就可以执行了。输出应该看起来类似图 12.2。你应该注意到，大部分测试都失败了，也就意味着在 sum 函数中有 bug。找到 bug，修复它，然后全部测试就应该通过。单元测试真是个大救星！

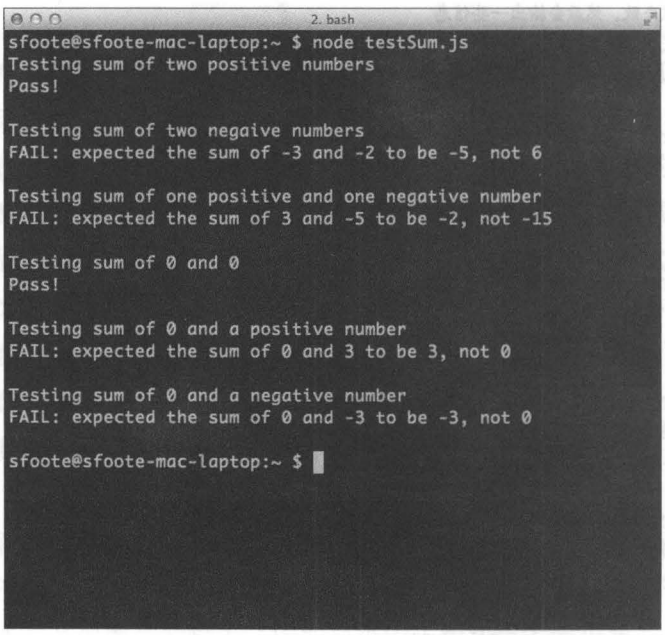

图 12.2　命令行中运行 testSum，显示有些测试没有通过

试着通读代码和注释，注释解释得相当清楚。这段代码只是用来演示我所谓的写代码来测试代码。实际上，你应该不会让你的真实代码（通常叫作应用代码）和测试代码放在同一个文件中。

注意看代码中所有的消息是如何描述的。首先，我要声明我要测什么，用

`console.log('Testing sum of two positive numbers');` 。然后，我用工具函数 test 运行测试。如果测试通过，我就把它打印出来，用 `console.log('Pass!\n')`。这里\n 是一个新的行，这会在每一个测试之后加入一些空白，让结果的可读性更好。如果测试没通过，我要打印出一条描述性的失败信息，其中包括输入、期望的输出，以及真实的输出。像这样的描述性信息让测试在失败的时候，更容易让人了解到底出了什么问题。

另一个要注意的地方，是怎么使用工具函数 `test`。我这么做是为了不用每条测试都重复写失败消息。真正的测试是在 `test` 函数内，所以我最好把它写对。这就引出一直以来我对于写自动化测试的一个困惑，如果我在写测试代码来测试我的应用代码，我要如何测试我的测试代码？我需要些测试代码来测试我的测试代码吗？到什么时候才算结束？我想到几个选择：

1. 在你开始写测试代码时，你必须穷尽一生不停地给测试写测试。
2. 一个测试都不写。
3. 将测试限定在三层深度。
4. 写测试，但保证他们足够简单，以至于不需要自动化测试。

选项 1 看起来并不有趣，也不太可能，所以其实这个并不是一个真正的可选项。选项 2 解决了无休无止写测试的问题，但是测试还是很有必要的，所以这个选项也不可取。选项 3 听起来像是《盗梦空间》的场景，而我尽量避免从电影中获取编程建议。最后就剩下选项 4，我觉得写简单测试是这个问题的正确答案。当你写测试代码时，你可以手工运行测试，确保测试代码如预期一样工作。如此说来，测试代码应该足够简单，以至于不需要自动化测试。

给 Kittenbook 配置测试

`testSum` 这个例子里还有些东西是我其实不太喜欢的。我觉得单元测试是好的，并且我觉得例子也很好地展示了单元测试中都应该包含哪些部分。问题是，`testSum` 在介绍单元测试的基本知识上做得不错，但并没有给你展示如何测试真正的代码。给 `sum` 函数写的单元测试可能不难，但怎么给像 kittenbook 这样的真实程序写单元测试呢？

第一步是以单元的方式思考 kittenbook 的代码。整个 kittenbook 项目要比 `sum` 复杂太多了，但每一个单元（函数）可能是一样简单的。事实上，这是写单元测试的

优点之一：为了让代码单元可测，代码必须写成小巧可控的单元。写成小巧可控单元的代码要更加干净，更加有条理，并且通常更不容易出错。在我们讲 kittenbook 代码整理成函数之前，它几乎是不可测的。我们不会尝试一次测试所有 kittenbook 的代码，但我们会隔离每一个函数进行测试。

好消息是，grunt 可以帮助我们做单元测试，我们会使用一个叫作 Jasmine 的测试库。Jasmine 主要提供了测试工具函数，比如 `test`，还有很多其他的东西。要配置好 Jasmine，首先需要安装 `grunt-contrib-jasmine`（这会安装 Jasmine 及它的所有依赖）。然后你需要添加一个 Jasmine 任务到 `Gruntfile.js`，见清单 12.2 和 12.3。

清单 12.2　用命令行安装 `grunt-contrib-jasmine`

```
## 首先，导航到 kittenbook 文件夹下，运行下面的命令
~/project/kittenbook $ npm install grunt-contrib-jasmine --save-dev
```

清单 12.3　添加 Jasmine 任务到 `Gruntfile.js`

```
module.exports = function(grunt) {
  grunt.initConfig({

    ...

    jasmine: {
      test: {
        src: ['js/values.js', 'js/prompt.js', 'js/getImages.js',
              'js/replaceImages.js', 'js/main.js'],
        options: {
          specs: 'test/*.js'
        }
      }
    },

    ...

  });

  // 加载 Grunt 插件

  ...

  grunt.loadNpmTasks('grunt-contrib-jasmine');
```

```
  ...
};
```

清单 12.3 只展示了 `Gruntfile.js` 里新添加的部分，这个新的 Jasmine 任务有一些类似的选项。我们用 src 属性告诉 Jasmine 要到哪里去找源代码。我按照 `concat` 任务里的同样顺序列出文件，因为我发现这样在执行测试的时候会避免一些问题。之后，我们用 `spec` 属性（在 `options` 对象里面）告诉 Jasmine 到哪里去找测试文件。你需要创建一个名为 `test` 的新目录，这是我们要放所有测试文件的地方。在你做完这些之后，试着用命令行在 `kittenbook` 目录中运行 `grunt jasmine`。你应该会得到一个警告消息，先是像 “警告：没有执行任何测试，可能是配置错误？使用--force 继续。(Warning: No specs executed, is there a configuration error? Use --force to continue.)” 这样的内容。这意味着 Jasmine 配置正确了，但是 Jasmine 找不到任何可运行的测试文件。当你真正有可运行的测试时，这个警告就会消失。所以我们开始给 kittenbook 写第一个测试吧。

规格说明（Spec）

单元测试通常被称为 Spec（specification 的缩写），因为它们详细描述了软件能做什么，不能做什么。在上一章中，你学到了写规格说明是做计划的一部分。写单元测试是写规格说明的一种很好的方式。如果全部单元测试都通过了，软件就达到规格说明的要求了。

我想具有简单的输入和输出的函数是最容易测试的函数，所以我们先以一个这样的函数开始：`validatePhoneNumber` 接受一个字符串作为输入，然后输出一个布尔值，如清单 12.4 所示。你还可以看到一些 Jasmine 工具函数的用法。这些函数在这里是提供帮助的，所以不要怕它们或觉得困惑。

- `describe` 函数描述了要测试的是什么。`describe` 接受两个参数：一个字符串，用来描述要测什么；一个函数，包含了被描述的测试。清单 12.4 种，你可以看到一个 `describe` 在另一个 `describe` 里，这完全没问题。外层的 `describe` 说我们要测试 `prompt.js` 中的函数，内层 `describe` 说我们要测试 `validatePhoneNumber` 函数。
- `it` 函数运行一个单独的测试。要是我的话，会起一个更符合语义的名字（比

如 test），可惜没有人问我的意见。it 接受两个参数：一个字符串，用来描述这个测试；一个函数，包含了测试代码。这个字符串通常以 should 开头，所以这个代码读起来就是“it should return a boolean（它应该返回一个布尔值）”。

- expect 函数检验期望。如果你的 expect 通过了，你的测试就通过了。如果你的 expect 失败，你的测试就失败了。如果你没有 expect，你的测试就永远是通过的，所以在 it 函数内你应该总是有至少一个 expect。expect 通常和另外的某个工具函数（比如 toBe）一起使用，所以 expect 这行代码读起来会像这样：“Expect [actual result] to be [expected result]（期望［真实值］是［期望结果］。）”如果真实结果不符合你的期望，测试就会失败。

清单 12.4 测试 validatePhoneNumber 返回布尔值

```
describe('prompt.js', function() {

  describe('validatePhoneNumber', function() {

    it('should return a boolean', function() {
      var result = validatePhoneNumber('23456');
      expect(typeof result).toBe('boolean');
    });

  });
});
```

清单 12.4 中的测试代码描述了 prompt.js 文件中的 validatePhoneNumber 函数。到目前为止，我们只写了一个测试，它声明了 validatePhoneNumber 应该返回一个布尔值。在测试中，validatePhoneNumber 被调用，而它的返回值被保存在 result 变量中。下一行说我们期望这个返回值的（数据）类型应该是一个布尔类型（typeof 是 JavaScript 中的一个操作符，返回操作数的数据类型）。注意，我们不关心 result 是 true 还是 false，只要它是布尔值就行。如果 result 的数据类型是一个布尔类型，测试通过；如果不是布尔类型，测试失败，而且我们知道一定是我们的应用代码出了什么问题。从命令行中运行测试，使用 grunt jasmine，然后你会发现我们的测试失败了。这是因为 validatePhoneNumber 使用字符串的 match 方法（返回一个数组或者 null）而不是正则表达式的 test

方法（返回一个布尔类型）。我们的单元测试帮我们找到了代码中的一个 bug。在 bug 修复之后，你应该会看到像图 12.3 一样的结果。

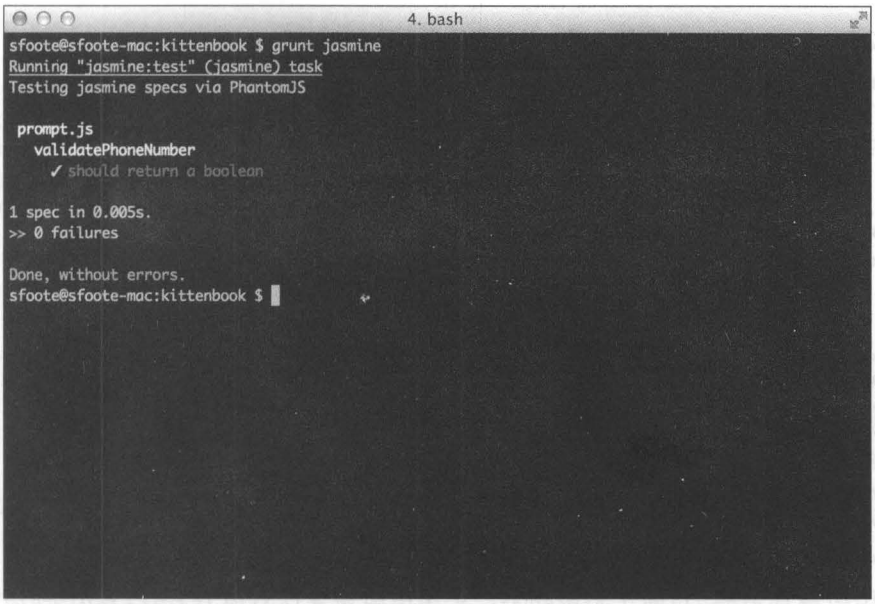

图 12.3 Jasmine 会在每个通过的测试前输出一个令人满意的绿色对号

失败时代

现在你可能认为一个好的测试是一个通过了的测试。测试失败不好，是吧？它们意味着什么东西坏了，而坏了就是不好的。我的观点是，你会希望测试在通过之前先失败。在我一开始写单元测试的时候，我非常享受。我爱死那些绿色的对号了，它们表明我的所有测试都是通过了的。我对自己非常自豪。我开始分享我的代码和测试给其他跟我一起工作的程序员们。然而，很快，我发现通过手工测试，我的应用代码有个非常严重的 bug。我的单元测试本来应该可以测出这个 bug，但由于某种原因，测试仍然是通过的。这说明我的测试写的太差了，以至于 `expect` 函数都没有被调用到，所以期望就一直没有被检测到，因此测试也就一直没有失败。现在，我经常先让测试失败，然后再让它们通过。如果我不能让测试失败，我就知道一定是出问题了。

我们来给 `validatePhoneNumber` 再写一个测试，并让它在通过之前先失败。我们想要测试 `validatePhoneNumber` 在输入一个合法电话号码时返回 `true`。我

们要添加另一个 it 函数到包含了 validatePhoneNumber 的 describe 里，如清单 12.5 所示。在你成功让这个测试失败之后，你可以更新代码，让它通过。

清单 12.5 Jenny，我知道你的号码，并且我确定它是合法的

```
describe('prompt.js', function() {

  describe('validatePhoneNumber', function() {

    it('should return a boolean', function() {
      var result = validatePhoneNumber('23456');
      expect(typeof result).toBe('boolean');
    });

    it('should return true when given a 1-800 number', function() {
      var result = validatePhoneNumber('1-800-867-5309');

      //修改下面的代码
      //
      // 我们知道它肯定会失败
      //
      expect(result).toBe(false);
    });
  });
});
```

间谍喜欢我们（我们也喜欢间谍）

在测试没有输入和输出的函数时，你没办法验证输出就是你期望的值，所以你通常就要测试这个代码做了你期望它做的事。举个例子，main 函数不接受输入，也不返回输出，所以我们要测试 main 函数调用了所有我希望它调用的函数。在 Jasmine 中你可以通过“监视（spy）”一个函数来检验函数是否被调用。监视函数可以让你掌握函数是否被调用，调用了多少次，调用时的参数是什么等。监视在应用代码中没什么太大用，但对于单元测试来说非常有用。我们来测试 main 函数，用 spy 监视那些它应该调用的函数，见清单 12.6。

清单 12.6 做间谍太酷了

```
describe('main.js', function() {
  describe('main', function() {

    it('should call getUserName', function() {
```

```
    spyOn(window, 'getUserName');
    main();
    expect(getUserName).toHaveBeenCalled();
  });

  it('should call getPhoneNumber with the value from getUserName', function() {
    spyOn(window, 'getUserName').and.returnValue('Jimmy');
    spyOn(window, 'getPhoneNumber');
    main();
    expect(getPhoneNumber).toHaveBeenCalled();
    expect(getPhoneNumber.calls.mostRecent().args[0]).toBe('Jimmy');
  });
 });
});
```

清单 12.6 中的第一个测试展示了监视的一个简单例子。spyOn 函数创建了这个监视。spyOn 的第一个参数是包含着函数的对象。getUserName 函数在全局作用域中，而在 JavaScript 中，所有全局函数都是 window 对象的一部分。第二个参数是要监视的函数名（以字符串的形式）。当监视配置好之后，我们可以调用 main 函数。最后，检验我们的期望：expect(getUserName).toHaveBeenCalled(); 。toHaveBeenCalled 在监视对象被调用至少一次的情况下通过，在没有被调用的情况下失败。

第二个测试展示了一个单元测试的重要概念：每个测试应该只测一件事。在第二个测试中，我们要测试 getPhoneNumber 被调用时传入的参数，是 getUserName 的返回值，但我们不想要真正去测试 getUserName。我们想要在一个测试中只测一件事，因为我们要明确知道如果测试失败，到底是哪里出问题了。如果我在 main 测试中测试了 getUserName 的某个部分，我就不知道到底问题是出在 getUserName 还是 main。我可以用 Jasmine 的监视功能强制 getUserName 返回特定值（这里是'Jimmy'），这样就确保了我只是在测试 main 中的功能。在测试代码调用 main 之后，我期望在最近一次调用 getPhoneNumber 时，第一个参数（索引为 0）是'Jimmy'：expect(getPhoneNumber.calls.mostRecent().args[0]).toBe('Jimmy'); 。

现在你明白单元测试是怎么回事了（还有 Jasmine 是怎么回事），你可以去给 kittenbook 写更多的测试。想好要测什么以及怎么测，在一开始可能挺困难，所以如果你卡住了，你可以参考我在本书网站上发布的代码中，第 12 章下面的单元测试。

写单元测试可能相当困难，但它绝对值得投资：记住，我们在写 kittenbook 单元测试时，在 `validatePhoneNumber` 中找到了一个 bug。当所有测试都写好后，运行 `grunt jasmine` 应该会返回给你一个类似图 12.4 的结果。

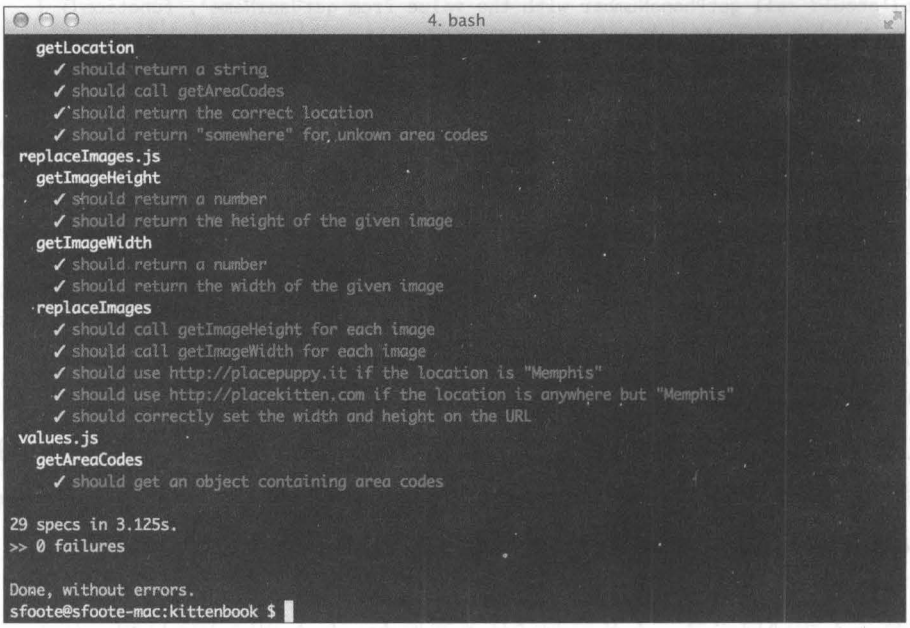

图 12.4　我一共给 kittenbook 写了 29 个测试（或者“Spec”），全部通过

测试驱动开发

如果写单元测试如此有价值，而且可以找到 bug，为什么我们要等到写完代码之后才开始写单元测试呢？一种写代码的方式就主张测试应该在应用代码之前写完。这个方法，通常被称为测试驱动开发，可能一开始看起来很疯狂，但实际上相当有效。写单元测试时先确保你的代码写成条理清晰的单元。如前面提到的，写测试很像是写规格说明。如果你先写测试，就确保你在开始写应用代码之前做了适当的计划。必须承认，在你写任何代码之前就想好你的代码应该做什么确实比较难，但如果你可以学会这种思考方式，测试驱动开发会提供很多好处。

集成测试

我有三个兄弟和两个姐妹。在一个有 8 个人（6 个孩子，两个家长）的家庭中成

长教会我，即使我们所有孩子在单独待着的时候都和天使一样，在我们 6 个挤在旅行车后座中的时候，我们也不会表现得规规矩矩。即使你的单元测试都通过了，当所有单元放在一起工作时，你还是可能会遇到问题。集成测试会测试你的整个程序，在所有单元一起工作的情况下。我喜欢把集成测试看成是自动的手工测试。就像 Jasmine 对于单元测试一样，软件可以帮你给你的软件创建集成测试。这类软件让计算机表现得像人一样，执行类似点击和打字这样的操作。一些机器人甚至真的会在智能手机上触摸和滑动，来自动化测试移动设备。集成测试通常是软件准备就绪，提供给用户之前的最后一步——最后一道防线。编写自动化集成测试超出了本书的范围，但如果你有兴趣学习更多知识，可以在网上找到很多资料。

尽早发现问题

做所有这些测试的最重要的原因之一，就是你要尽早发现问题。如果你在写代码的时候找到了一个 bug，你可以很容易修复它，甚至在任何人知道之前。如果你直到执行集成测试的时候才发现这个 bug，你可能已经写了很多代码，依赖这个有 bug 的代码。要修复这个 bug，你可能必须修改你的代码，以及依赖你代码的代码。如果你在一个已经打包并交付给用户的软件中发现了 bug，你就会冒着失去用户的风险。可能最差的情况就是用户开始依赖这个软件中的 bug 了，你修复了那些 bug，用户会很不开心（如图 12.5）。

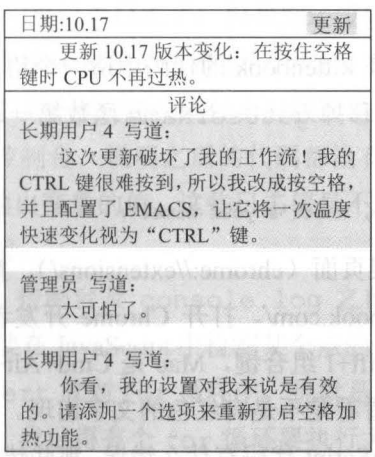

图 12.5 你的用户可能不会依赖你的 bug，但如果他们真的这么做了，如果你修复了这个 bug，他们会疯掉（卡通授权 xkcd.com）

调试

调试是指当你的程序工作不正常时，你尝试找出如何修复问题的过程。从我自己的经验看，调试可能是编程最好或者最坏的部分之一。如果知道如何使用调试工具，并且你知道要找什么东西，调试其实可能很有趣。然而，如果你只是知道程序坏掉了，而不知道如何修复它，调试可能会让人沮丧抓狂。

我最初的调试尝试就让人沮丧和抓狂。在我开始用 Perl 编程时，我知道的唯一运行代码的方式，就是在 Windows 文件管理器中双击我要运行的文件。这会打开一个命令提示符，这个窗口只有在代码执行期间才会保持打开（通常少于半秒钟）。在我的代码工作不正常时，我只能在窗口关闭之前瞥到一眼命令提示符上出现的一大堆输出。我不知道这个输出要跟我说什么。在我意识到我的代码出了什么问题时，我必须通读代码的每一行（用记事本，还记得吗？），尝试找出一个落下的分号或者没有闭合的括号。像这样的调试，感觉就像是可笑的猜谜游戏。这是很烧脑的工作，而我这么做，只是因为我并不知道有更好的方法。幸运的是，确实有更好的方法。

面对 Web 应用的时候，可以用浏览器的开发者工具来做调试。在第 5 章“数据（类型）、数据（结构）、数据（库）”中，你学习了 Chrome 开发者工具的控制台 console 是调试 JavaScript 的好地方。现在我们要从控制台 console 入手，深入了解如何使用开发者工具的其他功能。

错误

为了演示目的，我们在 kittenbook 的代码中造一个错误看看。打开 `release` 文件夹下的 `main.js` 文件，移掉 `getUserName` 函数第一行代码的右括号，第一行应该看起来像这样：

```
var userName = prompt('Hello, what\'s your name?';
```

现在打开 Chrome 扩展页面（chrome://extensions/），重新加载 kittenbook 扩展；然后访问 https://www.facebook.com/。打开 Chrome 开发者工具（Windows 和 Linux 上的键盘快捷键是 Ctrl+Shift+I 组合键，Mac 是 Cmd+Alt+I 组合键）。在开发者工具中，选择控制台标签页，看看是不是有红色的文字出现。你应该会看到像图 12.6 中展示的样子。如果你的 JavaScript 代码有什么错误，那些错误会在控制台中展示出来。必须承认，错误信息看起来不太清楚。错误通常不会告诉你到底哪里出问题了，但

如果你知道怎么看错误信息的话，它会告诉你找到一个错误，在代码的哪一行。

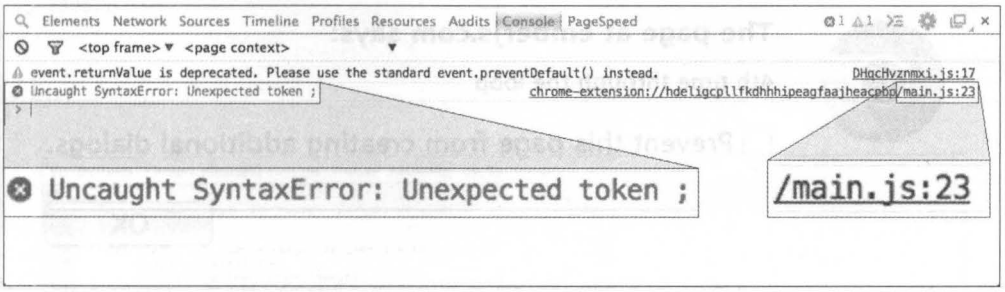

图 12.6 看起来像咒语，但其实错误可以告诉我们很多信息

图 12.6 中的错误在告诉我们，`main.js` 的第 23 行有问题。这个例子里的错误信息是有帮助的，尽管它并没有告诉我们到底出了什么错。这个消息告诉我们一个分号（;）出现在了 JavaScript 运行时觉得不应该出现的地方。这个例子的第 23 行，运行时本来希望在遇到分号之前会遇到一个右括号。错误信息并没有告诉我们整个事情的来龙去脉，但他给了我们足够的提示，是出了什么问题；至少，它告诉我们问题出在哪一行。当出问题的时候，第一件事要做的就是检查控制台上的错误信息。

日志

如果出了问题，你没有在控制台看到错误信息，还可以试试使用日志。我们已经用过几次 `console.log` 来记录日志了，但我们还没以调试为目的使用它们。一般来说，日志包含了程序在执行到各个位置的程序输出信息。如果什么东西看起来不工作，`console.log` 可以帮你判断代码什么时候被执行。当你运行程序的时候，如果日志信息没有出现在控制台，说明之后的代码都没有被执行。你还可以用日志在程序执行的特定点上，输出变量的值。这些日志信息不会真正修复你的 bug，但它们可以帮你找到 bug 的原因。

在我发现开发者工具、控制台及 `console.log` 之前，我会用更基本形式的日志：`alert` 函数。每次我想要在 JavaScript 中检测某个点的变量值，我就放一个 `alert` 在代码中。有时我会把 `alert` 放到 `for` 循环中，也就是说，弹窗会在每次 for 循环执行循环体的时候弹出来。我记得有个 `for` 循环要执行大概 100 次，那我就必须单击 100 次回车（见图 12.7）。使用 `alert` 并不是很好玩的日志方法。它确实有效，但幸运的是，你还有更好的办法。

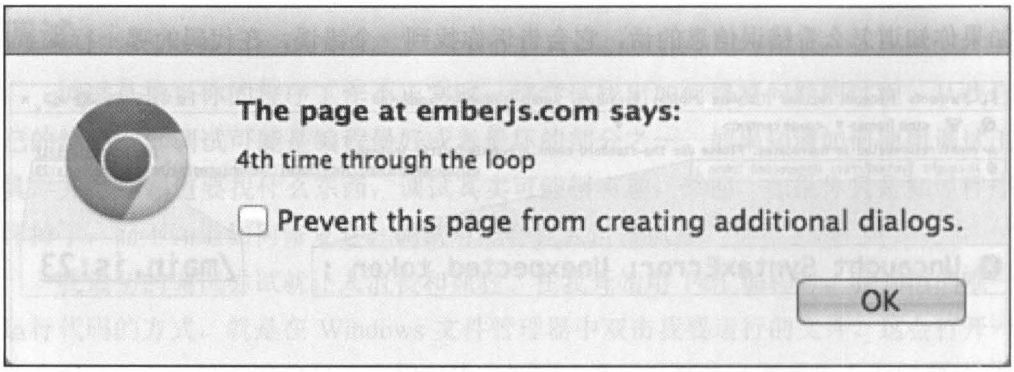

图 12.7　“阻止页面在此弹出对话框”选项在我用 alert 调试的时候还没有。要是当时有，可能会帮我节省很多时间

断点

错误信息和日志都很有用，但最强大的调试方法其实是断点。设置断点可以让你的代码在程序的某个特定位置暂停执行。在代码被暂停的时候，你可以查看所有变量的当前值。你可以一条一条执行断点之后的指令，看看是哪些指令会被执行，变量的值是怎么变化的。断点可以让你放慢代码的执行速度，准确看看到底发生了什么。

使用 Chrome 开发者工具，你可以通过两种方式设置断点。第一种，也是最简单的一种方式，就是在代码中放一个 debugger 语句。debugger 对其他代码没有任何影响，但如果开发者工具是打开的，它可以让代码停在那一行。debugger 很有用，但你需要确保在打包代码发给用户之前，移除所有 debugger 语句。代码中的断点可能会对你来说很有用，但对于用户就会很烦人。清单 12.7 展示了包含一个断点的 main.js，图 12.8 展示了代码在开发者工具中，停在了那一行。

清单 12.7　使用 debugger 暂停代码执行

```
function main() {
  debugger;
  var userName = getUserName();
  var phoneNumber = getPhoneNumber(userName);
  var location = getLocation(phoneNumber);
  var images = getImages();
  replaceImages(images, location);
  setInterval(function() {
```

```
    images = getImages();
    replaceImages(images, location);
  }, 3000);
}

main();
```

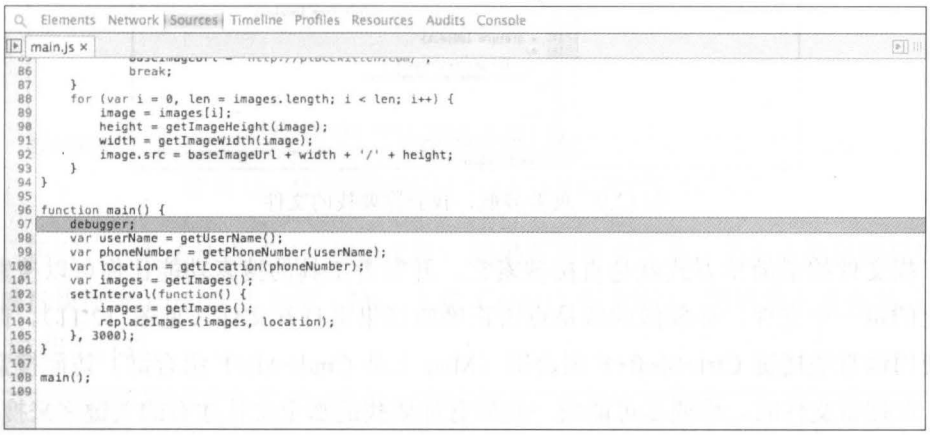

图 12.8 用 debugger 暂停代码执行

第二种设置断点的方式是在开发者工具中实现。点击源代码（Source）标签页，找到你要放置断点的那一行，并点击那一行旁边的行号（见图 12.13），找到你要设置断点的那一行可能是最难的部分，特别是你所在的页面有大量的JavaScript文件（比如 Facebook）。你可以通过点击展开图标（见图 12.9），在列表中找到对应的文件。当页面有很多 JavaScript 文件时，找到你的文件可能会比较困难（特别是当你在写扩展的时候——扩展的 JavaScript 文件是在源代码标签页下面的内容脚本（Content script）子标签页中）。

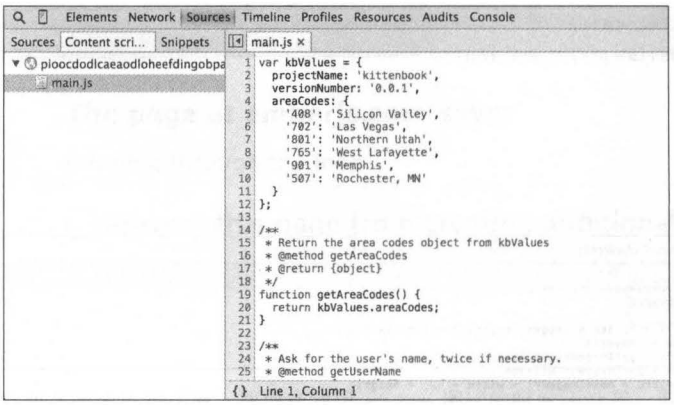

图 12.9　展开导航，找到你要找的文件

找文件的最简单方式就是直接搜索它。开发者工具的搜索功能让你可以搜索页面上的每一个文件，你要做的就是点击正确的结果来打开文件（见图 12.11）。你可以使用键盘快捷键 Ctrl+Shift+F 组合键（Mac 上是 Cmd+Alt+F 组合键）访问搜索功能。在搜索文件时，你要尽可能用一些只有你要找的那个文件才有的关键字来搜索。要把这个功能用在 kittenbook 上，你可能需要开启开发者工具的设置，让它可以搜索内容脚本（或者也可以说是扩展脚本）（见图 12.10）。你可以点击开发者工具右上角的齿轮打开配置菜单。

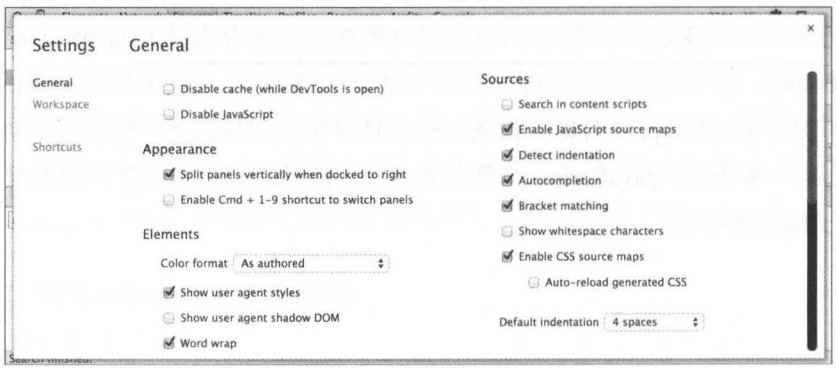

图 12.10　开启搜索内容脚本

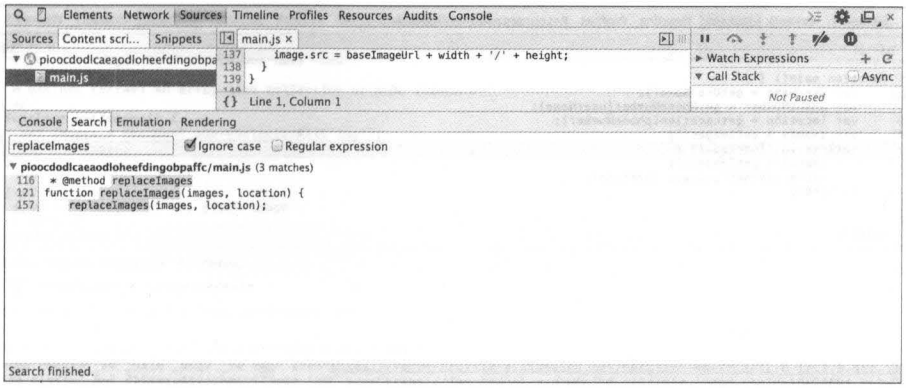

图 12.11 使用搜索源代码可以更容易的找到你要的文件

还有一个方法是通过名字搜索文件。如果要搜索文件名，先点击源代码标签页，用键盘快捷键 Ctrl+O 组合键（Mac 上是 Cmd+O 组合键），然后输入文件名（见图 12.12 和 12.13）。

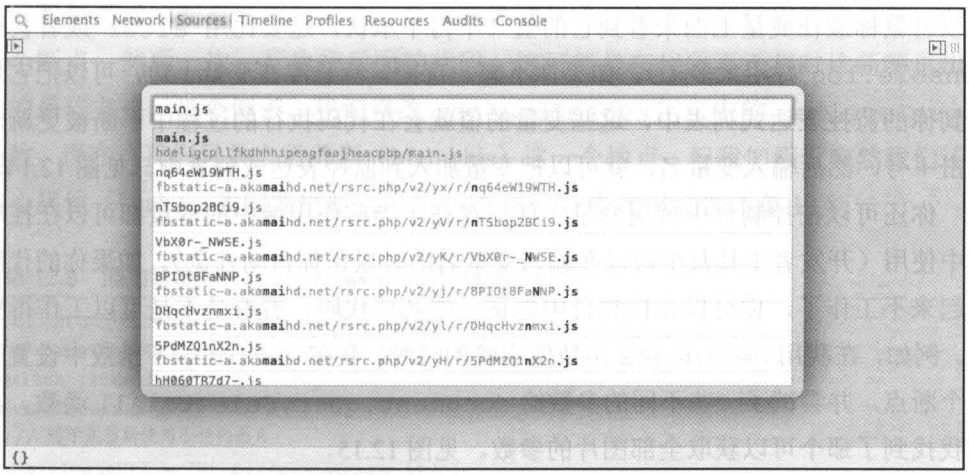

图 12.12 让 Chrome 为你做这件事：通过名字打开文件

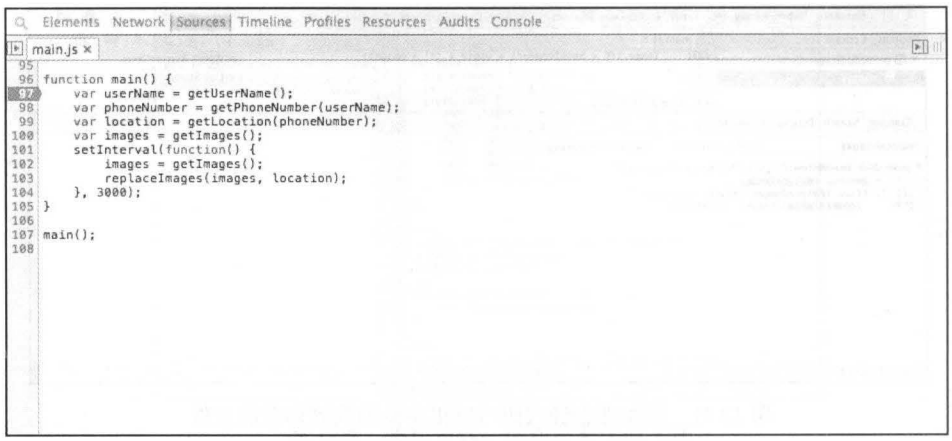

图 12.13　点击行号 97 在第 97 行设置断点

查看、监控和控制台

在你的代码在某个断点处暂停执行时，你可以做一些非常酷的事情。首先，你可以将鼠标放在变量上面来看到它的值（不得不承认，这要比用 `alter` 或者甚至 `console.log` 好用太多了）。如果你不想一直把鼠标悬浮在变量上边，可以把它加入到你的监控表达式列表中，这些变量的值就会在代码执行的过程中不断被更新。点击＋号，然后输入变量名，就可以把变量加入到监控表达式列表中。见图 12.14。

你还可以在控制台中使用变量。任何存在于当前作用域中的变量都可以在控制台中使用（开发者工具甚至可以在输入变量名的时候帮你自动补全）。如果你的代码看起来不工作了，你可以在控制台中尝试一些其他代码，看看是不是可以工作得更好。例如，在我用 `getImages` 函数做实验的时候，我在 `getImages` 函数中设置了一个断点，并尝试了一些不同的参数给 `document.querySelectorAll` 函数，直到我找到了那个可以获取全部图片的参数。见图 12.15。

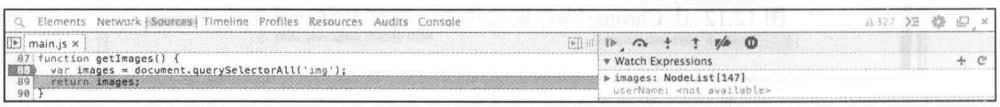

图 12.14　监控表达式让你立即看到想要关注的变量更新

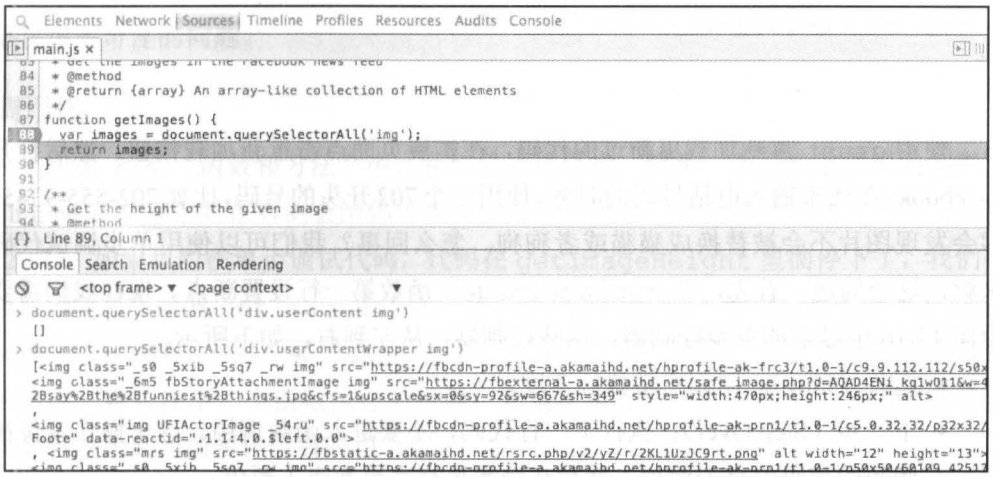

图 12.15 在控制台中使用当前作用域中的变量运行代码

单步执行代码

我最喜欢的断点使用方法之一是单步调试代码。单步调试代码的意思是，设置一个断点，然后一次一行执行后面的代码。这可能是在代码不正常时诊断哪里出问题的最佳方式。举个例子，我们修改 `replaceImages.js` 中的 `replaceImages` 函数。现在，我们只关心孟菲斯（Memphis）这一个地点，但我们假设突然我们还要关心犹他（Utah）。我们要更新 `replaceImage`，看起来就像清单 12.8。

清单 12.8 添加犹他到 `replaceImages`，但破坏了代码

```
function replaceImages(images, location) {
 var baseImageUrl, height, width, image;
 switch (location) {
 case 'Memphis':
  // 对于孟菲斯使用小狗的图片
  baseImageUrl = 'http://placepuppy.it/';
  break;
 case 'Utah':
  // 其他地方使用小猫的图片
  baseImageUrl = 'http://placekitten.com/';
  break;
 }
 for (var i=0,len=images.length; i&lt;len; i++) {
  image = images[i];
  height = getImageHeight(image);
  width = getImageWidth(image);
```

```
    image.src = baseImageUrl + width + '/' + height;
  }
}
```

使用 grunt 重新加载更新过的代码，在扩展页面点击重新加载按钮，然后打开 Facebook。在要求输入电话号码的时候，使用一个 702 开头的号码，比如 702-555-9345。你会发现图片不会被替换成猫猫或者狗狗。怎么回事？我们可以使用单步调试代码来解决这个问题。首先，在 replaceImages 函数第一行设置断点。现在我们需要用图 12.16 中显示的单步控制器，这些控制器，从左到右，如下所示。

- 运行（Play）：回到正常执行速度。
- 下一步（Step Over）：执行下一行代码，不要进入函数调用。这个操作就像在函数的代码中点击运行，然后在函数执行完再次点击暂停。
- 单步进入（Step Into）：执行下一行代码，如果下一行代码是函数调用，进入要调用的函数。
- 单步跳出（Step Out）：跳出当前函数。这个操作就像按下运行，直到当前函数完成执行然后在函数调用的这一行按下暂停。
- 禁用断点（Disable breakpoints）：当这个选项被选中，Chrome 会忽略掉你的断点
- 遇到错误暂停（Pause on errors）：当这个选项被选中，Chrome 会在发生错误的时候暂停执行

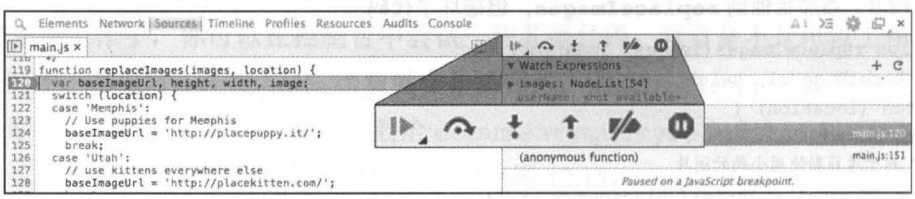

图 12.16　单步进入、单步跳出和下一步

通过不断地单步进入，可以进入到 replaceImages 函数，我们看到程序执行跳过了 switch 声明的所有条件。因此，baseImageUrl 没有被赋值，而因为 baseImageUrl 没有被赋值，浏览器不知道从哪里去找图片。在修改代码的时候，我们忘记了在 switch 声明中增加 default 条件。所以当地址不是孟菲斯（Memphis）或者犹他（Utah）的时候，程序就会中断。单步执行代码可以帮我们

解决很多奇怪的问题。

调用栈

在第 8 章“函数和方法”中，我们见过调用栈，就是一个函数被调用的列表。Chrome 开发者工具在右侧会给你可视化的展示调用栈，就在监控表达式下面（见图 12.17）。如果我们在单步调试代码，代码在 `getImageHeight` 里面停下了，我们可以在调用栈中看到 `getImageHeight` 被 `replaceImages` 调用，`replaceImages` 又被 `main` 调用，`main` 被（`anonymous function`）调用。你可以点击任意函数的名字，看看在下一个函数执行的时候，它是在执行哪一行。这个工具用来查看代码如何进入到当前暂停位置时很有用。

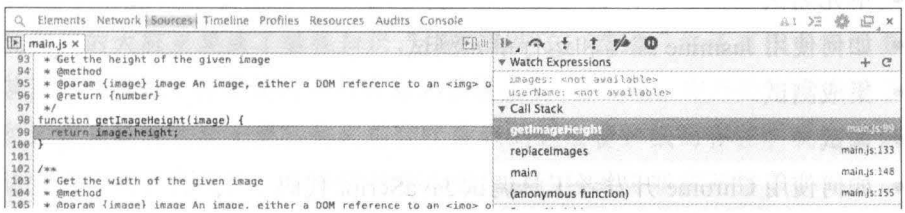

图 12.17 Chrome 开发者工具给你一个可视化展示调用栈的工具

找到根本原因

调试不是件容易的事。你真的要表现得像个侦探一样，仔细检查所有线索，才能发现为什么程序并没有按照预期的方式工作。在你排查故障的时候，你会经常找到很多表面症状，它们会诱导你去修复这些表面症状。我自己就干了好几次这样的事，哪一次都没有好结果。持续挖掘，直到你找到问题的根本原因——当你修复那个根本原因后，你就会一次修复所有表面症状。你修复了表面症状而不是根本原因的一个征兆是，这个修复让你的代码看起来更乱，更难维护。例如，如果你必须给一个函数添加参数，你可能是在修复表面症状，而不是真正的问题。

编码、测试、调试、不断重复

在你写代码的时候，你应该在工作流程中包含测试和调试。先写一点代码，接着运行测试，继续调试失败的测试，然后写更多的代码。当你写完足够多的代码后，你可以通过真正使用你的程序来做些集成测试（手工或者自动）。你可以调试在集成

测试时找到的问题，然后重复整个循环。当你找到 bug 时，创建一个新的自动化测试，确保 bug 不会再次出现。在你的工作流程中包含测试和调试会帮助你更快的写出高质量代码。

总结

直到我开始编程几年之后我才学会测试，而我第一次调试的尝试相当原始（还记得 `alert` 吗？）。你刚刚学到的测试和调试技巧会帮助你更快成为一个更好的程序员。本章你学到了：

- 手工测试
- 单元测试
- 如何使用 Jasmine 编写和运行单元测试
- 集成测试
- 调试
- 如何使用 Chrome 开发者工具调试 JavaScript 代码

下一章中，我将授人以渔，而不是授人以鱼。换句话说，你会学到如何在读完本书之后持续学习编程。具体来说，你会学到：

- 如何使用 Web 搜索编程问题的答案
- 去哪里提问，如何提问
- 如何通过教别人来学习

13

授人以渔：如何用一生学习编程

注

项目：写一篇博客，介绍你到目前为止学到了什么，并注册 StackOverflow。

现在你的大脑里装满了编程知识，快要爆炸了。然而，你还有很多东西要学，本书甚至无法覆盖（甚至任何一本书都无法覆盖）。要我说，你只是刚刚戳破了一层窗户纸，但从某种意义上说，你之前仅仅意识到这层窗户纸的存在。我这么说不是为了让你灰心或者打压你，本书到目前为止，你已经做得非常不错了，你应该对自己感到自豪。你已经为自己打好了地基，你应该知道，现在你可以盖起高楼大厦了。

有了这样的基础，你就可以更快地处理编程信息，更快地学习新概念。例如，当你听到朋友谈论他们在项目中使用的构建工具，你会知道他们在说的是什么。你已经知道什么是构建工具，它们的目的是什么，它们能做什么。如果你的朋友没有使用 Grunt，你就有机会学习一些构建工具，而你所学的知识将会很容易地融入你已经构建的基础之中。真正的挑战已经不像在第一次学习并理解它们时那么困难。后面的内容会教你找到你需要的信息。

如何搜索

让我为你画一幅图。假如你在做一个 JavaScript 项目，其中你得到了一个字符串格式的日期（比如，`'2014-10-08'`），你想要计算下一天的日期是什么。你已经学到过可以用 `parseInt` 将字符串转换成文字，所以你就用了这个函数（如清单 13.1 所示）。一切顺利。然后有一天你意识到你的程序产生了一些很奇怪的结果，但只是在老版本 IE 浏览器才出现：你的程序认为`'2014-10-08'`的下一天是`'2014-10-01'`，而不是`'2014-10-09'`。你设置了一些断点并单步调试代码，

你会发现 parseInt 看起来坏了：当参数是'08'时，在大部分浏览器中你会得到 8，但在 IE 中你会得到 0（对此的解释详见后面的注释）。你发现了这个问题，但你不知道如何修复它。现在你要怎么办呢？随着你不断地编程，你会不断地遇到类似的场景。比困境更加让人沮丧的是不知道如何脱离困境。

清单 13.1　使用 parseInt 将字符串转换成数字

```
function nextDay(str) {
  var num = parseInt(str);
  return num + 1;
}

var date = '2014-10-08';
var dateParts = date.split('-');
alert(nextDay(dateParts[1]));
```

在 Google 时代，每个人都是搜索专家。不到几秒钟，你就可以找到各种信息，比如你正在看的电影女演员的八卦信息。因为互联网以前和现在都是由程序员搭建的，互联网上关于编程的信息格外的多。当你遇到像前面 parseInt 这样的问题时，你肯定会找到答案，只要你知道在哪里找以及怎么找。

> **parseInt 解释**
>
> 为什么老版本 IE 的 parseInt 会做如此奇怪的事情呢？原因是 parseInt 也对十进制之外的数字有效（比如二进制、八进制，以及十六进制）。如果你不告诉 parseInt 你在使用什么进制，parseInt 就用它认为最合理的进制来解析数字，而'08'看起来像是个 8 进制数。新的浏览器倾向于默认使用 10 进制，即使字符串以 0 开头，但老版本 IE 仍然认为'08'是个 8 进制。不管怎样，你应该总是在使用 parseInt 时指定进制：parseInt('08', 10);（以 10 为基，或者十进制）会在所有浏览器中返回 8。

找到正确的关键字

如果你不知道要搜索什么，你就不可能找到结果。在第一次我要用转义字符时就遇到了这个问题。我不知道有种叫做转义字符的东西。我所知道的就是撇号在破坏我的字符串。在 parseInt 例子中，你可能并不知道哪里出问题了，那么你要如何搜索一个解决方案呢？关键在于找到正确的搜索关键字。如果你在找一些针对某

个编程语言的东西，就在搜索中包含这个编程语言的名字。然后想象其他遇到同样问题的人会如何描述这个问题。例如，在搜索 `parseInt` 问题的结果时，一个好的查询可能是“javascript parseInt returns 0”。当我运行这个搜索时，第一条记录就返回了我要找的答案。

如果你不确定要搜索什么，Google 的搜索建议可以帮助你。在你输入查询的开始部分时，提示可以帮助引导你找到可以搜到最佳结果的查询关键字。这些建议意味着其他人搜索过这些关键字并且得到了最佳结果，所以你也许能成功。图 13.1 展示了在搜索 `parseInt` 解决方案时，一些有用的（以及一些不那么有用的）建议。

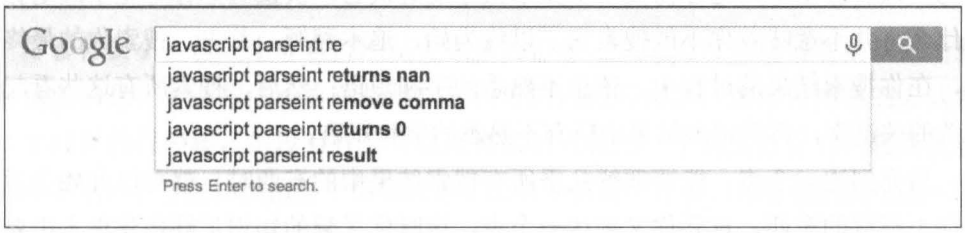

图 13.1 谷歌的建议可以帮你理解你要搜索的东西

升级

你要找的资源依赖于你处于学习的哪一个级别。比如，如果你在考虑学习写 Python 程序，你想要找些关于用 Python 的好处的介绍性信息。在这样的例子中，你会想要用类似“why use Python”（为什么使用 Python）和“Python features”（Python 特性）这样的搜索关键字。在你决定 Python 适合你之后，你会想要学习 Python 基础。这时，你要搜索一些类似“Python tutorial”（Python 教程）这样的东西。在你用 Python 工作一段时间后，你也许会遇到一个问题，需要使用正则表达式。你已经知道什么是正则表达式，以及在什么时候要使用它；你只是需要知道如何在 Python 中使用它。那你可能就会搜索“Python regular expression”（Python 正则表达式）或者“Python regular expression documentation”（Python 正则表达式文档），找到关于 Python 中正则表达式如何工作的文档说明。现在假设文档抽象，难以理解；你可以搜索“Python regular expression tutorial”（Python 正则表达式教程）或者“Python regular expression example”（Python 正则表达式示例），找到一些比较容易消化理解的 Python 正则表达式信息。这些例子要说明的就是，如果你在搜索关键字中指定你处于哪个级别，你就更容易找到需要的信息。

错误

错误可能让人特别困扰，因为你的代码不工作，同时错误信息看起来又特别模糊。不过错误信息有一个好处：错误信息中的文字基本上总是不变的。这就意味着，其他人也见过相同错误信息，并且很可能已经找到了解决方案。当你遇到一个不理解的错误信息，复制粘贴整个信息到谷歌，一般你都会找到问题的答案。

以终为始

在一开始，你也许并不知道自己到底要搜索什么。你可能不完全理解要解决的问题，或者甚至不知道你需要学多少东西才能解决它。仅仅因为你不完全确定你要找什么，并不意味着你不能搜索它。以终为始，追本溯源。首先，搜索你的最终目标，在你搜索结果的过程中，留意不熟悉的字和短语；然后，搜索所有这些看起来相关的关键字，再次记录结果中所有不熟悉的字和短语。

当你到达一个点，能够理解几乎所有搜索结果中的东西时，就可以开始学习那些你不理解的东西，直到你又到达一个点，这时你了解的知识足够搭建你本来要搭建的东西。这个过程会帮助你在达到目标的过程中学到很多很棒的知识，或者，至少帮你意识到你的目标也许需要比预期更多的工作和学习才能达成。

这种追本溯源式的搜索引导我学习了 Web 编程。我之前提到过给我的第一个网站实现“自动补全”功能。那时我并不知道那是自动补全，所以我搜索了一些类似“google search suggestion”的内容，这让我发现了关键字“autocomplete”。我读了一些关于 autocomplete 的文章，确保那就是我要找的东西，并且发现了关键字 AJAX。我了解到 AJAX 就是实现基于用户输入的文字展现搜索推荐的技术。我做了些 AJAX 调研，发现你必须写 JavaScript 才能使用 AJAX。所以我开始学习 JavaScript，直到我学了足够的知识，可以使用 AJAX，我就可以搭建自动补全系统了。在这个过程中，我学到了很多 HTTP、Web 服务器，以及数据结构的知识。我相信用追本溯源的方式解决编程问题会有给你很大的回报。

识别高质量资源

不幸的是，互联网上不是什么东西质量都很高。编程信息和其他内容都是如此。小心那些质量不高的网站和书。因为相信网站上那些过时、写得不好的信息，我学到很多错误指导和坏习惯。不幸的是，在能够区分高低质量资源之前，你也许得吃

几次亏。如果你访问的网站有一大堆广告，特别是它们跟编程基本没关系或者一点关系都没有，那你就要小心了。小心那些尝试把各种话题都囊括进来的网站（比如，About.com 包括了报税、园艺、编程，以及发型设计）。当你见到那种说你要学的东西很简单的声明，也要小心，如果太简单了，你很可能学到的是错误方式。大部分编程语言、库和框架都有网站，包含完整的、高质量的文档。这些网站一般包含完整的入门信息、初学指导、教程，以及参考材料。从这些网站上找到的信息通常都是高质量而且正确的。

个人博客：隐藏的宝藏

尽管你要小心从一个非官方网站上得来的信息，你还是可以在博客上找到一些有价值的信息。专业程序员喜欢把他们解决过的问题写出来，通常以博客的方式发表。文档和教程很棒，但没有什么可以跟真实的人描述一个真实世界的问题及其解法相提并论。不过在你相信这个信息之前，还是要调研下作者。看看他们的 LinkedIn 简历、StackOverflow 档案（见下一小节，“StackOverflow”）、GitHub 档案（见第 15 章“高级主题”）、Twitter 档案，以及任何其他你能找到的资源。如果他们看起来明白他们在说什么，你就找到了一个很好的信息源。这类博客就像编程导师；他们给你如何更好编程的免费建议。记住，如果你接收他们的建议，要在文档中提一下那篇博客。

什么地方、什么时候，以及怎么问编程问题

自学编程是个困难的事情，因为你可能找不到人来帮你解决编程问题。幸运的是，我们有互联网，所以尽管你个人并不认识任何程序员，你还是可以向真人提问编程问题，并且得到真实的（好）答案。获得问题答案的关键在于在什么地方、什么时间，以及如何问问题。

什么地方

找到正确的地方问编程问题可能和问对问题一样重要。如果你跟错误的人问了正确的问题，也许得到了答案，但你不会得到一个好答案。因为你想要一个好答案，就需要知道在哪里问问题。

打电话给朋友

如果你是因为没人教你、指导你或者引导你而自学编程，那就要改变一下了。事实上，你可以一个人走得很远，但你如果有个朋友可以帮你指点正确方向，你会学得更多更快。有时候，你已经认识一两个程序员，他们可能会很高兴回答你的问题（特别是现在你有了基础，你可以问出好问题）。如果你不认识任何程序员，就去认识一个。多亏了像 www.meetup.com/这样的网站，你可以找到很多在你这个领域的程序员会议。找一个那样的群组，认识些人，跟他们学学。

问真人问题是学习和寻找答案的最高效方法。一个真人可以看你的代码，帮你调试，以及帮你找到解决方案。一个真人可以给你量身定制的帮助，因为那个人知道你的背景，以及你当前的知识等级。因为他已经回答了你的其他问题，并且熟悉你工作的项目。朋友可以教你和帮你找到自己的答案，而不仅仅是给你针对你当前问题的答案。如果有一点点可能，一定要问朋友。

StackOverflow

即使你有程序员朋友，他不可能回答你所有的问题。所以 StackOverflow（http://stackoverflow.com/）是一个很棒的资源。StackOverflow 是一个在线的程序员问答社区。网站上的每个东西都通过点数来评分。如果你问了一个问题，很多人都觉得有用，那些人会给你的问题投赞成票（见图 13.2）。如果你对一个问题提供了很好的答案，人们会给你的答案投赞成票（见图 13.3）。相反，不好的问题和不好的答案会得到反对票。最好的问题在搜索结果中会获得高优先级，而最好的答案会被显示在页面的最顶部。声望点会奖励给（或者移除）用户的每一个赞成/反对票，累积的声望值还会显示在页面上所有的用户名旁边。所有这些投票和点数的背后理念，是要让好东西浮现出来，而不好的东西逐渐隐藏起来。这个系统非常好；你几乎可以找到任何编程问题的答案。如果问题还没有被问过，你可以问一下，通常可以在一小时内得到答案。

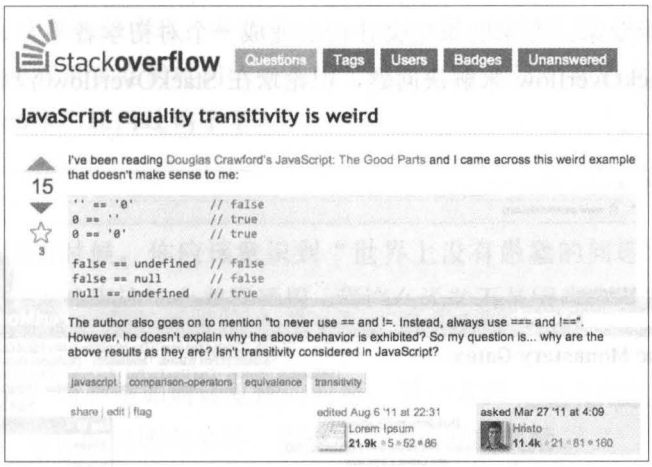

图 13.2 一个 StackOverflow 上的问题，有投票总数，以及提问者的名字和声望值

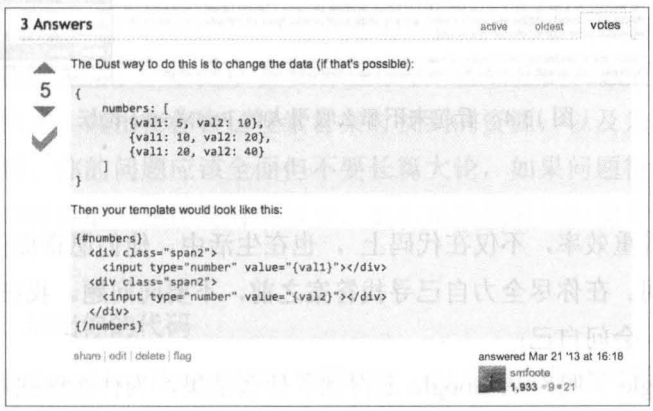

图 13.3 StackOverflow 上的一个答案，答案会根据投票数排序，“√”表明这是个“被接受的答案”

论坛

在我刚开始编程的时候，在线论坛是当时找到编程问题答案的最佳方式。在我学 Perl 的过程中，我在 Perl 论坛花了很多时间（见图 13.4）。尽管我很感激这些论坛，我还是很高兴 StackOverflow 及其他类似的问答排名网站很大程度上取代了论坛。你知道，论坛很难做信息筛选。一个问题可能会得到好多页的回答，而问题的最佳答案可能在任意一页，或者可能根本就没有最佳答案。要想知道答案的唯一方式，就是通读每一页的每一条回复。如你在图 13.4 中看到的，论坛通常都不怎么好看。然而，论坛的一个优势是它们一般是针对某个特定的编程语言，因此，论坛会吸引这

门语言的世界级专家。专家的聚集会让论坛变成一个对初学者不友好的地方。一般我建议就用 StackOverflow 来解决问题，但论坛在 StackOverflow 没解决的情况下可能会有帮助。

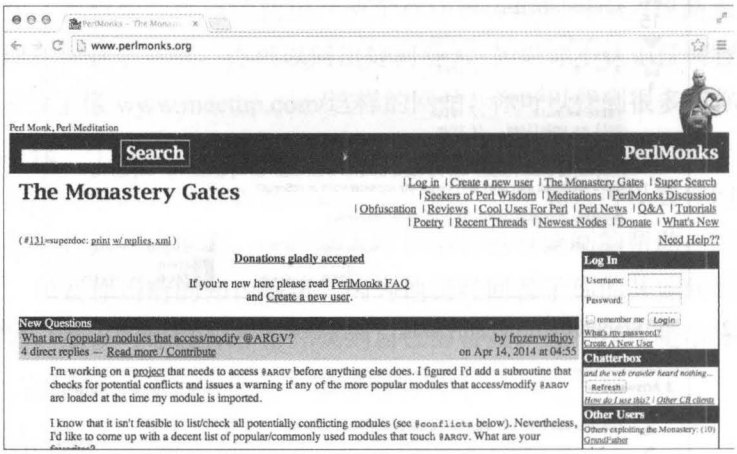

图 13.4　看起来不那么吸引人的 PerlMonks 论坛

什么时候

程序员很看重效率，不仅在代码上 ，也在生活中。他们愿意提供帮助，但也会珍惜他们的时间。在你尽全力自己寻找答案之前，不要问问题。我在 StackOverflow 上问问题之前，会问自己：

1．我 Google 了吗？从 Google 上得到了什么结果，为什么这些结果没有回答我的问题？

2．我读过文档了吗？我读了文档的哪一节，为什么文档没有回答我的问题？

3．有人问过同样的问题吗？为什么之前问题的答案在我这个场景不能工作？

作为程序员，我也看中效率。我想要自己找答案，但我不想没有目的的搜索，所以如果我已经试过了 Google，试过了文档，也试过了搜索 StackOverflow，还是找不到答案，那我知道，是时候问一个问题了。

玩具鸭

通过问问题，你可以经常找到答案。不断地尝试用简洁的语言描述问题，可以帮你更好理解你的问题，也许甚至可以揭示问题的答案。找一个毛绒玩具（鸭

> 子就不错），放在电脑旁边，当你遇到问题的时候，问问鸭子。玩具鸭是一个好的倾听者，而且有很多空闲时间。如果问了鸭子之后你还没有搞清楚，去问一个真人吧。告诉他你已经问过鸭子了。

怎么问

当要问问题的时候，你应该意识到“世界上没有愚蠢的问题”这种说法在像StackOverflow这种编程论坛上并不适用。我这么说并不是因为刻薄，而是让你知道，如果你问了一个坏问题，你可能会被上课而不是得到答案。社区成员会专注于如何修复你的坏问题，而不是如何修复你的问题。另一方面，如果你问了个好问题，你会得到好答案，看你的问题有多好，你也许还能得到些奖励。

尽量简单

在StackOverflow和其他论坛上回答问题的人都是志愿者，他们不会因为回答你的问题而得到补偿（除了声望点），所以你应该尽可能让你的问题更容易回答。写一个清晰的问题描述，列出在你自己搜索答案时找到的资源，以及为什么那些资源没有回答你的问题。你的问题应该全面但不要长篇大论，如果问题很长，你应该加一个概述，读者理解你的问题所花时间越少，他们用来找解决方案和写答案的时间就越多。

不要害羞：贴出你的代码

我第一次在StackOverflow上问问题的时候，没有包含我的代码，因为我不太好意思。我怕人们会取笑我的代码，而不是回答我的问题。事实上，在没看到导致问题的代码时，编程问题是很难回答的。这就像是医生用电话进行诊断，“听起来你有胃肠感冒，或者也许是阑尾炎。”贴出跟你问题相关的代码会帮助你得到更好的答案。

通过教别人来学习

通过做项目，你可以学到很多，但你可以通过教别人学到更多东西。在你学习如何编程的过程中，正是最适合教别人如何编程的时候。通过把刚学到的东西及你正在学的东西教给别人，刚好是你以一个初学者需要的视角理解这些知识。老程序员们也许比你知道得更多，但那也意味着他们解释问题的方式会很复杂，而且难以理解。这么看来，你要比其他人更适合教初学者。

回答问题

你不需要找一个编程老师的工作来开始教学。你可以通过回答问题开始。在StackOverflow上找找你能回答的问题。你已经了解了一些关于HTML、JavaScript，以及Google Chrome扩展的知识，你可以尝试回答跟这些相关主题的问题。如果你不习惯在StackOverflow上回答问题，你还是可以回答问题。找其他在学编程的人（或者说服一个朋友学习编程），然后你们可以互相回答问题。你们两个会比各自独立学习学到的东西更多。

写博客

写作也是个学习的好方法（我猜这就是为什么你上学时要写那么多论文的原因）。你已经学习了很多可以写得好内容，比如JavaScript的函数如何工作，源代码是什么，以及为什么要使用Grunt和如何使用Grunt，如何使用Chrome开发者工具，以及如何创建Chrome扩展。在你开始写这些主题时，你会发现你的理解是有缺口的。在你尝试填补这些缺口时，你会更深刻地理解这个概念。记住，你具有教别人的最佳角度，因为你自己正在学习这个概念。如果你不习惯写一个给全世界看的博客，你可以给你的玩具鸭写封信，它是个很好的聆听者。

总结

本章你学到了一些重要的工具，能够帮你在已经学到的知识基础上继续添砖加瓦。具体来说，你学到了：

- 搜索答案
- 问好问题
- 通过教别人来学习

下一章中，你会学到：

- 在读完本书之后该做什么
- 如何开始你自己的项目
- 在线教育工具，免费的和收费的

14 构建你的技能

注

项目：注册 Udacity、Coursera、codeacademy，或者类似的网站，并开始上一门课，选择下一本要读的书。

你做到了！你已经为自己打下了坚实的编程基础，而且在这个过程中你创建了一个很酷的 Chrome 扩展。你知道如何解决自己的问题，而且你知道在遇到问题时要去哪里找答案。那么现在你该做什么呢？有了你现有的技能和知识，你已经可以搭建一些强大的东西。所以去搭建一些很棒的东西吧。然后持续学习，这样你可以构建更棒的东西。你的旅程刚刚开始。

做你自己的 kittenbook

在 kittenbook 上你已经做得很不错，但项目显然没有结束。那两个弹出窗口，向用户请求输入名字和电话号码比较烦人，而且它们不应该在每次用户加载 Facebook 页面时都弹出来（名字和电话号码不会经常变化，那为什么要不停地问呢？）。还有很多功能可以添加。如果用户想要猫和狗的混合图片呢？如果用户什么图片都不想看呢？我故意将 Kittenbook 留作一个未完成的状态，这样你可以通过添加你想要的功能，让这个项目变成你自己的项目。你已经熟悉 kittenbook 代码，以及它们如何一起工作，那么你就具备了扩展它的良好前提。玩得开心点，把 kittenbook 做成你自己的。

给 Facebook 重新设计风格

你有没有想过 Facebook 用绿色会比用蓝色更好看（见图 14.1）？你有没有曾经希望右边的聊天侧边栏更宽一点？你有没有对那些图标应该长什么样子有自己的想

法？用扩展添加一些自定义的 CSS，你可以实现上面说的所有改变。现在你有了修改 Facebook，或者任何你想要改的网站的力量。不过要记住，要遵守你正在修改的网站的服务协定，不要做有害的事。把你的力量用在正确的地方。

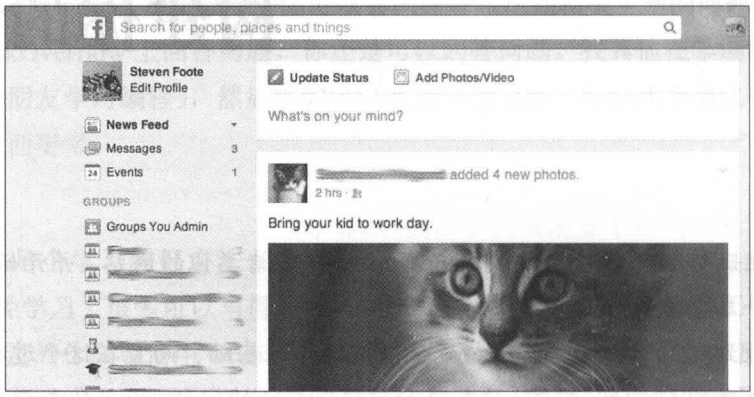

图 14.1　好吧，可能 Facebook 确实用蓝色更好看一些。至少我们试过了。

添加新功能

除了修改 Facebook 的外观，你还可以做更多事情。你还可以给 Facebook 添加新功能。你想要直接在 Facebook 里看到朋友的最新推特（Twitter）或者兴趣志（Pinterest）更新？只需要花一点点时间，你就可以学习 Facebook、Twitter，以及 Pinterest 的 API，并给 Facebook 添加全新功能。

kittenbook 选择用 Facebook 的决定其实是随机做出的，我们原本可以用任何喜欢的网站。事实上，你仍然可以用任何喜欢的网站。通过修改 `manifest.json` 中的 `content_scripts` 数组，你可以给其他网站添加可爱的猫猫狗狗，可能性是无限的。清单 14.1 展示了如何让你的 JavaScript 运行在 www.pinterest.com，但你必须对 `getImages.js` 做些修改，才能让 Pinterest 上的图片被替换掉（图 14.2 展示了 kittenbook 运行在 Pinterest 上的效果）。

清单 14.1　一大堆猫猫狗狗侵入 Pinterest

```
{
  "manifest_version": 2,
  "name": "kittenbook",
  "description": "Replace all the pictures on facebook with pictures of kittens or puppies",
  "version": "0.0.1",
```

```
"content_scripts": [
  {
    "matches": ["*://www.facebook.com/*", "*://www.pinterest.com/*"],
    "js": ["main.js"]
  }
]
}
```

图 14.2 猫的 Pinterest。你能看出区别么？

分享你的 kittenbook 专属版本

当一切准备就绪，向你的朋友展示你的 kittenbook 专属版本。教他们如何在他们自己的计算机上安装（可能还要教他们怎么禁用它）。如果你做了什么真正有价值的东西，你甚至可以通过在 Chrome 在线商店（https://chrome.google.com/webstore/category/apps）发布你的扩展，把它分享给全世界。

找到你自己的项目

如果在读完本书后，我只能给你一个建议，那应该是启动你自己的项目。我希

望你觉得 kittenbook 项目很有意思，即使它有点傻。Kittenbook 只是个娱乐项目，用来帮你学习不同的编程概念而已。最终产品对你或者其他任何人没什么用。但现在你学完了所有这些概念，你有了启动一个真正有用的项目工具。你有没有一直想要某个 iPhone 应用？你有没有想象过一个能够简化你和你同事生活的工具？发挥你的创造力，把世界建设得更好一点。

解决你自己的问题

如果你在解决自己的问题，你的项目就会简单很多。可能你的问题是你不得不一遍一遍重复做同样的日常工作（记得第 3 章“认识你的计算机”里面重命名几百个文件的例子吗？）。也许你的问题是你有个电子游戏的好创意，但还没人做过。在我做会计的时候，我的问题是我真的不擅长管理我的小时数（对会计来说这是个必须具备的技能），所以我做了图 14.3 里面展示的 Chrome 扩展。没有人比你更了解你的问题，而你对别人问题的理解也不会比对自己的问题更深，所以最适合解决的问题就是你自己的问题。即使没有别的人想用你的软件，这个工作还是值得的，因为它对你有用。假如说，你遇到了一个问题，其他和你类似的人可能有同样的问题，那么其他人可能会从你的工作中受益。不论如何，在搭建软件的过程中，最适合解决的问题就是你自己的问题。

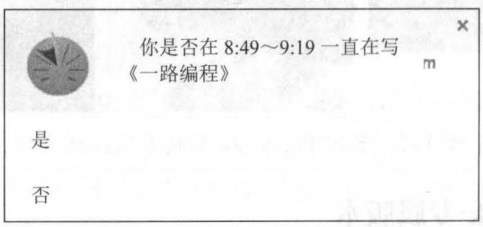

图 14.3　因为我太不擅长跟踪自己的小时数，我做了个 Chrome 扩展，每 15 分钟给我一个提示。最后，几百个同事和陌生人开始用我为自己做的这个工具

志存高远

当你选择了一个项目，要目标高远。保持学习的最好方式就是做一个有点野心勃勃的项目，即使你知道你还没有足够的技能搭建这个项目。当你已经开始一个项目，你对其充满热情，不论是不好的文档还是设计难看的论坛都不会阻挡你实现这个项目的步伐。我的第一个大项目是一个网站；通过那个网站，我学到了如何使用

Linux、HTTP、关系型数据库、PHP、HTML、CSS，以及 JavaScript。那些技能最终让我在 LinkedIn（硅谷的主要科技公司之一）找到了一个程序员的工作。如果你已经知道如何搭建你想要的东西，你就不太会成长或者学到太多，而且你肯定也不会觉得太有意思。

获得帮助，提供帮助

如果你有机会，找个朋友和你一起为你的项目工作。实际上，和朋友一起工作你会比全部自己做学到更多东西。如果你对要做什么项目还没有想法，你可以帮你的朋友做他的项目。对于事情应该怎么做，你和你的朋友会有不同想法，而且你们可以互相学习。可能和别人一起合作的最重要的原因，就是学习如何和别人一起在软件项目上工作。编程也许看起来像是个孤单的工作，但其实，编程在团队协作的情况下会做得更好。在软件项目中合作是一种技能，而且是很重要的技能。

> **做个网站**
>
> 我喜欢网络，如果你在想做个应用，我强烈推荐你使用网络作为平台。最容易分享软件的方式就是给别人一个你的网站链接。网络上有很多不可思议的东西，而你已经具备了创造这些不可思议的能力。另外，网络也很有意思。

开源项目

你已经知道了使用其他人的工作来加速自己的工作。通过给开源项目做贡献，你可以让你的工作被别人所用。给开源项目做贡献是个很棒的经历，而且通过跟才华横溢的程序员一起工作，你可以学到很多。你已经熟悉一些开源项目，包括 Google Chrome、Mozilla Firefox 和 Linux。如果你做了些 JavaScript 的调研，你可能已经听说过开源的 JavaScript 库，比如 jQuery 和 underscore。甚至编程语言，像 Perl、Python 和 Ruby 也是开源的。世界在开源软件上运行，而你有机会成为其中的一部分。

GitHub

GitHub 是一家为软件项目的协作提供出色工具的公司。GitHub 对开源软件的使用免费。这一优惠，加上高质量的工具，意味着 GitHub 已经成为新的开源软件项目事实上的归宿。很多流行的开源项目都托管在 GitHub，所以如果你想要寻找开源项

目来做贡献，GitHub 是个好的起点。其他还有些类似 GitHub 的网站，包括 Google Code 和 BitBucket，不过 GitHub 是最大的开源项目集合。

找项目

对我来说，找到一个我可以做贡献的开源项目是最难的部分。你想找个项目，让你可以做些有意义的贡献，或者甚至什么贡献都行。这意味着像 Mozilla Firefox、Google Chrome 和 Linux 可能看起来有点不在这个范围。你也许可以从贡献给项目中实际用到的那些开源库开始。Grunt 插件可能是一个比较合适的起步项目，因为这些项目的规模都相对比较小。小项目要更容易开始，因为代码更少，文档比较容易读完。这类项目大部分都有个 GitHub 页面的链接，在那里你可以找到代码和如何做贡献的介绍。找到 Fork Me 字样的 GitHub 丝带，见图 14.4。

图 14.4　表明开源项目在寻找贡献者的丝带

贡献的不同方式

通过添加或者修改代码来做贡献也许刚开始会有些困难和让人望而生畏，但这并不是唯一的贡献方式。你还有其他贡献方式，所以你可以在真正改变项目之前，先熟悉一下项目和代码。例如，你可以考虑下面几种选项：

- 寻找和定位 bug
- 添加和更新文档和用户指南
- 寻找和修复文档和代码注释中的文字错误以及其他错误
- 在 StackOverflow 上回答关于项目的问题

当你通过这几种方式对项目更加熟悉之后，你会对真正的修改代码感觉更自然，而项目的负责人会更愿意接收你的代码修改。从小处着手，慢慢成长。

创建你自己的项目

关于开源项目，最棒的一点在于每个人都可以创建一个，包括你。事实上，我强烈建议你创建一个自己的开源项目，即使你觉得没人会用它。可能没人会用到它，但创建一个开源项目会教你很多重要的东西。那些看起来很重要的东西，比如代码质量、结构、文档和测试，在你把代码发布给全世界看的时候会变得更加重要。谁知道呢？也许什么人找到你的项目，并且开始使用它，甚至更好，给它做贡献。

免费在线教育

很多网站都专注于教你如何写代码，其中很多都是免费的。有了目前的编程基础，你对充分利用这些资源做好了准备。这里我列出了一些我觉得最有用的资源，但还有很多其他的。我建议你浏览下面这些资源，自己找一些资源。

欧拉项目

如果你喜欢数学，即使只有一点点，那么欧拉项目（https://projecteuler.net）是一个非常好的资源。欧拉项目用数学来教计算机编程，并且同时使用计算机编程来教数学。欧拉项目包含了一系列挑战，需要数学知识和计算机编程知识一起来解决。这些挑战相互依赖；所以你可以使用上一个挑战中学到的技能来解决下一个挑战。如果你对编程的理解比较深刻，起初几个挑战的编程部分相对简单，但挑战很快开始变得相当困难。你可以使用任意你喜欢的编程语言来解决这些问题，但你肯定需要使用编程来解决它们。如果你对数学感兴趣，你一定要花些时间来解决欧拉项目的挑战。

> **解决数学挑战，获得工作**
>
> 2004 年，Google 有个好玩的招人点子。Google 想用解题的乐趣来吸引那些喜欢解决问题的程序员，所以他们打出一个广告牌，上面写着一个好玩的数学问题，和欧拉项目上的问题类似。这个问题是：去“{e 中第一次连续出现的 10 位质数}.com”（见图 14.5）。对数学和计算机编程的好奇心真的都可以满足，对通过解决广告牌挑战而得到 Google 工作的人来说尤其如此。

图 14.5　Google 聪明的招聘信息

Udacity

Udacity 上有不停增长的编程课程，由专家讲授，涵盖编程和计算机科学多个不同领域。大部分课程使用 Python 作为编程语言，但有些课使用 Java，JavaScript，以及其他语言。Udacity 课程是按照难度排序的，所以你可以从入门级课程开始，然后不断升级。课程被组织成很多短视频（最多大概 4 分钟），期间会带有小测验。大部分课程都是围绕某一类项目来讲，但学到的知识可以很容易地应用到课程和项目之外。我建议从下面这几个课程开始：

- Web 开发课程，教你如何搭建博客
- CS101 课程，教你计算机科学的基础，以及搜索引擎的工作原理

Coursera

Coursera 和 Udacity 类似，只有一点不同。这里的课程由不同大学的教授讲授，它们感觉更像是大学课程。这些课都有时间表，以及作业截止日期和课程设计。正因为如此，Coursera 的课会有些紧凑，但这也会帮你学到更多，而且更快。Coursera 提供很多学科的课程（而 Udacity 专注于计算机科学），而且大部分计算机课程适合更有经验的程序员。Coursera 也提供一些初级课程，而且即使你不完全理解高级课程的内容，你还是可以学到很多。我真的很喜欢上过的 Coursera 课程，但紧凑的时间表和截止日期确实更容易造成旷课和中途放弃。我建议从下面几门课开始：

- 大众编程（Programming for Everyone）

- 用 Python 介绍交互式编程（An Introduction to Interactive Programming in Python）

codeacademy

Codeacademy 会帮助你学习特定编程语言的编程基础，提供 HTML/CSS、JavaScript、PHP、Python，以及 Ruby 的教程。这些教程对于熟悉一门语言、看看是否想要投入时间进行完整学习很有用。codeacademy 课程提供了嵌入在浏览器内的文本编辑器，在这里你可以输入代码并可以看到结果，也就是说你不需要在你的计算机上安装编程语言的运行时环境。然而，它也意味着调试你的代码要更困难一点。另外，因为你没有安装运行时环境，你只有在 codeacademy 的网站中才能用上你学到的技能。我建议从下面几门课开始：

- JavaScript 课程——看看有多少是你已经学会的
- Python 课程——对比 JavaScript，学习它们的相似和不同

Khan Academy（可汗学院）

可汗学院是学习各种东西的好地方。最近，可汗学院发布了一个 JavaScript 介绍课程，教你如何用 JavaScript 画画和做动画。像 Udacity 和 codeacademy 一样，可汗学院的课程给你提供了内置文本编辑器来运行你的 JavaScript 代码。

教程

我前面提到的网站都是用来扩展通用知识的，但教程是用来学习如何完成某个特定任务的。举个例子，通过读 http://gruntjs.com/上的“入门教程”，我学会了如何创建 Grunt 任务。各种在线教程的质量参差不齐。我曾经看过一些教程，教我用很糟糕的方式来完成一些事情。因为读了一个教程，根据教程所说，我把很多数据放到了字符串中，而不是对象或者数组。这个故事的目的是让你小心对待来自陌生来源的教程。然而，一个好的教程可能帮你提升你的编程技能，到达更高层次。

付费教育

在你开始认真对待自己要学的东西时，你也许会考虑在学习方面进行一些投资。几乎所有你要了解的编程知识都是在线免费的，那么不论你选择什么付费教育，都

应该在你可以免费获得的东西之外，提供额外价值。

读书

在我刚开始学习编程时并没有读很多书。我发现所有能从书中找到的东西都可以在线上免费得到，那为什么我要为书付钱呢？我花了一年时间尝试理解如何正确的使用 Git，直到我找到了一本关于 Git 的书。在我读完这本书的一两周之后，我学到了关于 Git 我想知道的一切，而且发现了一些我甚至都没意识到我应该知道的东西。从那时起，我意识到编程书籍是多么有价值。尽管书中的所有信息都可以在网上免费找到，但信息并没有很好组织起来，而且网上还有很多不良信息。书的作者会把那些最终你也许可以从网上找到的信息以一致的描述方式组织起来，作者还会以正确的顺序展示各个概念，教给你那些你自己不会发现的东西。现在不论我要学习什么新技术，我首先会尝试找本书，我建议你也这么做。

Udacity 和 Coursera

Udacity 和 Coursera 都提供课程的付费版本。付费版本和免费版本的内容一样，但是他们在课程结束时提供认证证书。这些证书换不来大学文凭，但技术公司开始认这些证书了。如果你在找一个程序员的工作，你可以考虑付费版本。

Udacity 试着给你真实世界的体验，付费课程将因此更进一步。在 Udacity 付费课程中，除了在免费课程中得到的所有内容，还包括下面这些：

- 带有反馈和代码审查的课内项目
- 来自教练的个性化指导和学习速度
- 和教练的按需交流
- 课程完成的认证证书

Coursera 对每门课程以及每组专业课提供签名追踪。如果你付费听一门课，你会在完成课程之后收到一个认证证书。专业课是一系列具有同一主题的课程，以一个大作业作为结束。如果你成功完成了每一门课程，并且完成了大作业，你会得到由 Coursera 和创建该课程内容的大学一起颁发的专业课程认证。再次说明，这些认证不是大学文凭，但在简历中看起来会很不错。

Treehouse

Treehouse专注于Web设计和开发，其目标是教你找到工作的必备技能。Treehouse课程的全部材料都可以从Web的其他地方找到，但Treehouse课程会把所有这些信息管理和组织成合理的课程。你不用搜索整个互联网去找这些信息，也不需要想下一步要学什么；Treehouse都帮你做了。Treehouse的最大优势之一是课程内容的频繁更新。技术变化很快，所以Treehouse的每周更新可以确保你学的不是过时内容。

总结

你已经很好地完成了任务，做好继续做更棒事情的准备。你应该为此感到骄傲。本章你学到了：

- 启动你自己的项目
- 给开源项目做贡献
- 精进编程知识的在线资源

下一章是你接下来可能想要学习的内容的预览。你将学习：

- 设计模式
- 面向对象编程
- 版本控制
- Git

15
高级主题

注

项目：安装 Git，将你的 Chrome 扩展项目转到 Git 仓库。

你做到了！我要谢谢你，让我成为你学习编程旅程的一部分。你已经学了这么多，而且你准备好了继续学习。现在你可以和软件开发者聊天，并且能够理解他们在说什么（至少一部分）。你已经准备好拿起一本编程书或者开始学习一个在线编程教程，你知道它们在讲什么，而不是完全迷失和抓狂。更重要的是，你准备好了开始做自己的项目。你应该为你做到的这些感到骄傲。

现在你的脑袋里装满了新的编程术语和概念，你也许觉得别的东西已经放不下了。如果确实是这样，你现在可以合上本书，并带着当之无愧的成就感离开。这一章会介绍一些接下来可以考虑学习的高级编程主题，如果你读的话，这将是一个奖励。如果现在还不是开始高级主题的时机，我很理解；休息一下，过几天再回来。然而，如果你渴望学习更多东西，继续读下去。

版本控制

你已经学过了很多工具，有用来写程序的（文本编辑器和 IDE），用来编译和构建程序的（像 Grunt 这样的构建工具），以及用来调试程序的（比如 Chrome 开发者工具）。像这些一样，版本控制系统也是个重要的编程工具。版本控制系统是用来管理程序变化的工具。版本控制让你在项目每次修改源代码的时候做一个快照，每个快照都包含变化的描述和原因。一丝不苟的保持记录，即使在你不打算发布项目整个发展历程的情况下依然很有用。如果你的软件因为某种原因不工作了，版本控制让你在找问题的时候，很容易的回滚到上一个（可工作的）版本。

当项目在使用版本控制时，项目的所有代码都保存在一个中央位置，叫作仓库。要开始做项目，你必须从仓库“签出”代码的一个本地副本（本地意味着文件在你的计算机上）。你对本地代码的复制进行修改，当你的修改准备就绪，你可以“签入”到仓库中。签入意味着把你做的修改发给仓库，包括描述这些修改的信息。在签入的这个点上，你创建了一个新的项目快照。图 15.1 展示了这个过程是如何工作的。

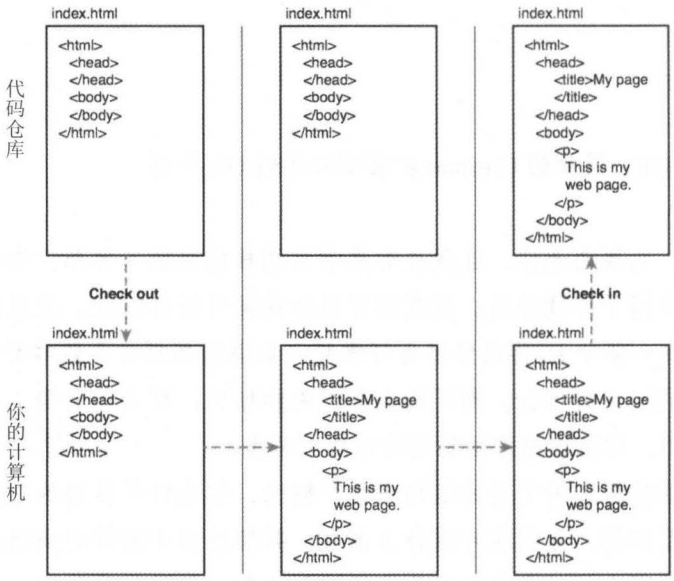

图 15.1　版本控制流程的简单可视化展示

为什么使用版本控制

除了追踪项目中的修改这个能力，使用版本控制的最有力原因之一是为了合作。如你在下一节中将要看见的，版本控制配置的方式让它成为各种规模团队的理想工具。版本控制还帮你以更专注的方式写代码。就像函数，每一次签入都应该有个特定目标。在一个版本控制系统中工作时，你应该只做跟当前签入相关的修改，这帮你保持注意力集中。最后，版本控制让你随意做实验，不用害怕破坏整个项目。回滚能力意味着即使你全都搞砸了，你可以很快回到正确的地方。除了回滚，分支的概念（见下面关于分支的小节）意味着你可以不用签入而进行试验。版本控制值得花时间去学习，不管你自己工作还是和团队一起工作。

和团队一起工作

你已经见到了如何通过签出和签入来跟仓库一起工作。签出和签入的过程允许很多人在同一时间，在同一个项目上高效工作，而且互不打扰。如果我们在一个项目上工作，我们会创建一个中央仓库，然后每次签出到本地副本。假如我们一起搭建一个网站，你要开始写 JavaScript，而我开始写 HTML。当你签入你的 JavaScript 修改时，仓库被更新，但我的代码副本还保持不变。当我签入我的 HTML 修改时，仓库被更新，但我还没有你的 JavaScript 修改，而你还没有我的 HTML 修改。要从仓库中获取最新的修改，我们各自都要想仓库请求“更新”。仓库用我们上次更新之后的代码修改，来更新我们的本地副本。图 15.2 展示了这个过程。

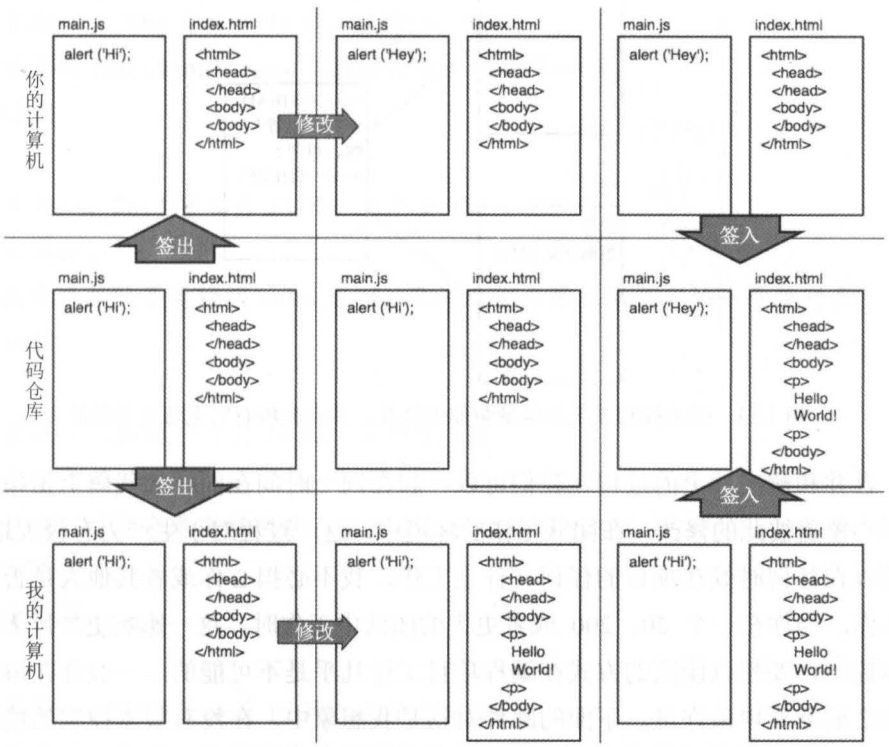

图 15.2 一系列签入和更新，保持每个人同步

现在我们假设你需要对我还在用的 HTML 文件做些修改。会发生什么呢？你对 HTML 文件做了修改，然后签入，然后我结束了我对 HTML 文件的修改，并尝试签

入。当我签入的时候，版本控制系统会注意到我的 HTML 文件版本是旧的，因为你在我上次更新之后签入了你的修改。我将不得不从仓库获取你的更新，然后仓库才能让我签入，这样是确保我签入的修改不会覆盖你签入的修改。这就是版本控制的魔法之所在。在我从仓库请求更新时，版本控制系统会尝试把你的更新合并到我 HTML 文件的本地副本中。如果发生冲突，版本控制系统会高亮冲突，并提示我，我必须在签入之前解决他们（见图 15.3）。举例来说，如果你和我都尝试修改一个文件的同一行，就会发生冲突。我可以通过把你的修改手工合并到我的文件中，这样就解决了这个冲突。

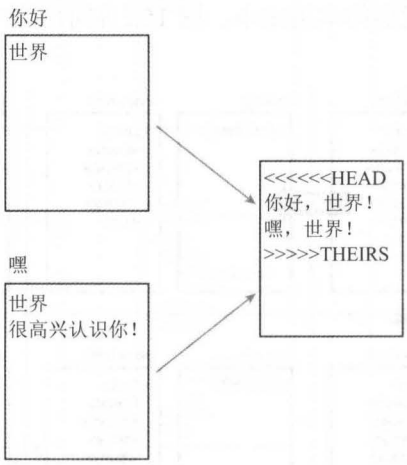

图 15.3　版本控制系统会尽量帮你做合并，并高亮所有它无法合并的冲突

合并和解决冲突的过程让我们可以一起在同一时间在同样的代码上工作，而不用担心覆盖彼此的修改。在团队工作的环境中，这一过程对于生产力有极大地影响。我可以在任何时候在项目的任何文件上工作，我不必担心你或者其他人是否要在上面工作。当你在一个 20、200 或者更大的团队中工作时，这一影响更加巨大。没有版本控制，要想以团队的方式在编程项目工作几乎是不可能的。一般你在编辑任意文件之前需要申请许可。下面的邮件对话是我想象中，在没有版本控制的情况下，合作软件开发可能的样子。很糟糕。

> 星期一，2014 年 4 月 14 日，上午 8：42
> 大家好，

周一愉快:P 我今天早上要开始对 index.html 做些修改，有没有人在编辑这个文件?

谢谢，

史蒂芬

星期一，2014 年 4 月 14 日，上午 8：50

史蒂芬，

我想杰瑞今天上午正在 index.html 上工作，你最好跟他确认下。

保罗

星期一，2014 年 4 月 14 日，上午 8：52

不要动 index.html！Jerry 正在修改它，我是下一个。

哈利

星期一，2014 年 4 月 14 日，上午 9：03

大家好，

我也在排队等着修改 index.html。也许我们应该在休息室搞一个签到表?

萨利

星期一，2014 年 4 月 14 日，上午 9：08

谢谢大家，

萨利，听起来是个好主意，你可以把我的名字也放上去吗?

谢谢，

史蒂芬

星期一，2014 年 4 月 15 日，上午 12：17

等会儿，各位。休息室中的签到表对我们这些在悉尼的人没什么用啊。我也要对 index.html 做修改。

莱恩

有了版本控制，我可以随时修改任何文件，让软件来接手合并的工作。如果有什么冲突，我可以在它们出现时处理掉。版本控制让团队合作更加高效。没有签到表，没有排队，没有邮件，只做你该做的，让版本控制来处理最难的部分。

> **代码评审**
>
> 以团队方式工作的最大优势之一是你可以从团队其他成员写的代码中学习，而且他们也可以从你的代码中学习。软件开发团队的一个常见的实践是代码审查，团队其他成员必须评审并批准你的代码修改，然后这些修改才能被签入。这些代码评审意见也许一开始看起来比较让人生畏，但它们对每一个参与其中的人可能非常有价值。项目的整体质量趋向于不断提高，团队的整体知识积累和专业度也一样在提升。创建 StackOverflow 的人建立了公开的代码审查网站 http://codereview.stackexchange.com/。不要害羞，主动让别人给你做代码审查吧！

Subversion

Subversion 是目前最受欢迎的版本控制系统之一。版本控制的基本特性在所有版本控制系统中都会支持，但每个系统的功能和实现都稍有不同。Subversion 是一个中心化版本控制系统，也就是说，Subversion 有个中心服务器，仓库就存在这个服务器中。每个开发者都可以随时从中心仓库签出某一时刻的代码副本，但只有服务器里保存了完整的版本历史。当你想要项目的历史信息时，你必须通过网络连接到一台运行着的中心服务器。因此，如果服务器宕机，整个系统就崩溃了。如果一个系统的某个部分会导致整个系统崩溃，这个部分就叫作单点故障点。单点故障点不是个好事情。保存中心仓库的服务器就是 Subversion 的单点故障点。不过，Subversion 仍然在大型项目（包括 Google Chrome）中是很流行的版本控制系统。只要采取预防措施，能够保障服务器的安全运行，Subversion 还是很好用的。还有些其他中心化版本控制系统，但 Subversion 是最流行的。

Git

Git 是我首选的版本控制系统。Git 的原创作者（利纳斯·托瓦兹，Linus Torvalds）也是 Linux 的原创作者，而且 Git 就是为了 Linux 开发团队而写的版本控制系统。Git 的设计初衷是要快，而且能够适应不同规模团队的不同规模项目。Git 还解决了中心化版本控制系统的单点故障问题。在 Git 中，每个计算机（不仅仅是服务器）都保存

完整的版本历史。如果服务器宕机，其他任意一台计算机都可以立即作为服务器使用。在讲 GitHub 的时候你已经了解了一点 Git，但现在我们来学习如何用它工作。

Clone

因为 Git 对于版本控制有不同的理念，它使用一套不同的术语。首先你不只是在一个仓库上“签出”代码副本；你用 Git 会得到代码及完整的版本历史。所以，Git 使用 clone（克隆）这个词，而不是“签出”，你的本地仓库是服务器仓库的完整克隆。使用命令 `git clone`，带上服务器仓库的 URL 作为参数，就可以在命令行克隆一个仓库（见清单 15.1）。

清单 15.1 使用 git clone 克隆

```
# 克隆我早期在 GitHub 上提交的一个仓库，一个时钟
sfoote@sfoote-mac:projects $ git clone https://github.com/smfoote/html5-clock.git
sfoote@sfoote-mac:projects $ cd html5-clock
sfoote@sfoote-mac:html5-clock $ # Now I'm ready to start making changes
```

Add

在克隆仓库之后，就可以开始对代码做修改了。Git 让你可以细粒度的控制如何跟踪你的修改。在你做了一些修改之后，你需要用 `git add [filename]` 来告诉 Git，哪些修改是你要跟踪的。只有那些被包含在 `git add` 命令中的修改才会被包含在下一个快照中（Git 中的快照叫做提交，commit）。如果你想在下次提交中包含全部修改，可以用 `git add -A`（`-A` 代表 all，是全部的意思）。注意，`git add` 不会向 Git 版本历史添加任何东西；它只是告诉 Git，哪些修改要包含在下一次提交中。你可以用 `git status` 查看哪些修改会被包含在下次提交中。见清单 15.2 中使用 `git add` 的例子。

清单 15.2 添加文件，为提交做准备

```
# 对 index.html 做些修改
sfoote@sfoote-mac:html5-clock $ vim index.html

# 确认哪些文件会被包含在下一次提交
sfoote@sfoote-mac:html5-clock $ git status

On branch master
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

# 把那些修改加入到下一次提交中
sfoote@sfoote-mac:html5-clock $ git add index.html
sfoote@sfoote-mac:html5-clock $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified: index.html
```

commit

在你准备好给你的项目做快照时，用 `git commit` 命令会向版本历史中添加一条记录，其中包含了所有用 `git add` 包含进来的修改。所有你做过的修改，但是没有用 `git add` 包含进来的，仍然在那里，但它们不会被提交。运行 git commit 添加到版本历史中的记录仅仅保存在你的本地仓库中。清单 15.3 是这个过程的示例命令，图 15.4 用图形展示了这个过程。如果你的项目只有你一个人，提交就是最后一步。然而，如果你以团队方式工作，就必须再做一步操作，把你的修改分享给项目中的其他人。

清单 15.3 提交一个修改及其提交信息

```
sfoote@sfoote-mac:html5-clock $ git status
On branch master
Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)

      modified: index.html

sfoote@sfoote-mac:html5-clock $ git commit -m "Fix typo in index.html"
[master 1234567] Fix typo in index.html
 1 file changed, 3 insertions(+), 2 deletions(-)
```

图 15.4 只有在已修改桶里的文件才能被移动到已添加桶，而只有在已添加桶中的文件才能移动到已提交桶

Pull 和 Push

到目前为止，我们运行的 Git 命令（除了 `git clone`）都不需要网络连接。这是 Git 强大功能之一；不论你在火车上、公交车上、飞机上还是其他任何没有互联网的地方，你都可以用 Git 仓库工作，而 Subversion 的几乎所有操作都要连接网络。不过，到了要把你的本地修改同步到服务器仓库时，你还是需要网络连接。在 Subversion 中，你要签入修改，但在 Git 中，你已经用 `git commit` 做好了快照，所以你要“推送”修改。要更新本地仓库，获取其他开发者做的修改，你要“拉取”。对于基本的 Git 使用，服务器仓库叫作原点（origin），当你推送和拉取时，你必须告诉 Git 服务器仓库的名字，见清单 15.4 中的示例。

清单 15.4 用 `git pull` 和 `git push` 同步服务器仓库

```
# 从服务器仓库中拉取更新
```

```
sfoote@sfoote-mac:html5-clock $ git pull origin master
From github.com:smfoote/html5-clock
 * branch master -> FETCH_HEAD
Updating 68125d1..4525da7
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

# 将我本地的提交推送到服务器仓库中
sfoote@sfoote-mac:html5-clock $ git push origin master
```

Git 工作流

学习和熟悉 Git 需要花一些时间。你也许会奇怪，为什么你要费这么大的劲去用 `git add`、`git commit`、`git push` 和 `pull`。看起来这些大量额外工作就是为了追踪代码变化，是不是？我来说一个版本控制系统存在的理由，而且是很有说服力的理由。直到我开始编程两年以后，我才知道版本控制这个东西。我写了两年的程序，没有用到版本控制，那么为什么我要花时间去学一些让人抓狂的签入、提交、更新和拉取呢？

有一天，我搭建的网站整个崩溃了。我不知道为什么，只有一个很模糊的想法，应该是我过去几天更新的代码有问题。我没办法回滚到之前的代码版本，因为我一个版本都没有。最后我找到并修复了这个问题，但我仍然不想学习版本控制。相反，我建了一个 Google doc 文档，每次对网站作出修改，我就写下日期、修改目的，以及改了哪些文件。换句话说，我在尝试重新创造一个版本控制系统，但我错过了版本控制系统能带给我的真正好处。当我最后学会了 Git，就再也不想回去了；那些收益值得你花大力气去学习添加、提交和推送。我现在在每个项目里都使用 Git（包括本书）。

我之前对 Git 保持观望的一个原因是我不是非常理解我到底该怎么使用它。我不希望你也有一样的问题，所以考虑下这个非常基本的 Git 工作流：

1. 对仓库中的代码做些修改。
2. 将修改添加到你要提交的修改集合中。
3. 使用 `git commit` 把 `git add` 添加的修改提交到仓库。
4. 用 `git pull origin master` 更新本地仓库，获取其他开发者做的修改。
5. 用 `git push origin master` 推送创建好的提交快照到服务器仓库。

在 kittenbook 项目中使用 Git

是时候检验下你的技能了。你已经学了很多遇到问题怎么找答案的知识，现在问题来了。kittenbook 需要版本控制，而你需要搞清楚如何配置。第一步是安装 Git（提示：GitHub 有一些很棒的资源，介绍如何开始用 Git）。第二步，在安装好 Git 之后，你需要把 kittenbook 目录转换到 Git 仓库（提示：这里只需要一条 Git 命令，但不是 `git clone`）。最后，你需要给项目创建一个初始快照，之后你就可以按照前面讲的 Git 工作流来工作了。

我意识到，给 kittenbook 配置 Git 这个任务看起来好像无谓地增加复杂性。在本书中我一直都在给你尽可能多的信息，因为我知道，如果一本书假设我知道一些实际上并不知道的信息，读起来会多么令人抓狂。但你已经准备好自己走下去，我知道你还不知道怎么安装 Git，但我还知道你有帮自己找到答案的工具。你会成功的，但如果你遇到阻碍，记得使用像 StackOverflow 或者有经验的朋友这样的资源。

分支

还有一个你应该知道的 Git 特性，那就是分支。Git 分支的概念一开始有些难以理解（至少对我是这样）。Git 分支是保持不同任务之间清晰的组织和隔离的一种方式。举个例子，假如你在给 kittenbook 添加新功能（例如，在 LinkedIn 上展示猫咪），但你有个进行中的任务，更新所有文档。没有分支的话，你就需要手动跟踪哪些修改是文档修改，而哪些修改是 LinkedIn 猫咪的修改；你肯定不想把一个未完成的功能跟文档修改一起提交，反之亦然。有了分支，你就可以创建一个叫作 `documentation` 的分支，在这里你对文档作出修改，还可以创建另一个叫作 `linkedin-kitten` 的分支，在这里你可以添加新功能。你可以很容易地在不同分支之间切换来做不同的任务。当你的修改都准备就绪后，你可以用 `git merge` 将这些分支合并进“master”分支。

这里介绍的 Git 显然不全面。Git 包含的内容要比我在这几页中介绍的多得多。如果你对深入学习 Git 感兴趣，我强烈推荐你看《Pro Git》这本书，由 Scott Chacon 所写。我曾经不觉得 Git 很好，也不理解 Git，直到遇到这本书。读过之后，我想把 Git 用在各个地方。这本书写得很好，对 Git 初学者很友好，而且可以在 http://git-scm.com/book 免费获取。

OOP（面向对象编程）

现在讲一点完全不同的东西，我们来聊聊面向对象编程（OOP）。你也许已经熟悉这个词了，不管是从以前的经验还是在读本书的过程中做的调研，但你也许没有真正理解它的意义。

首先，我认为 OOP 并没有看起来那么难懂，但它却是很强大。OOP 是一种很搞笑的组织和共享代码的方式，因此，代码变得更容易写了。面向对象编程这个名字来自于这样的理念，程序应该基于对象搭建（对象的类型参考第 5 章《数据（类型）、数据（结构）、数据（库）》）。我发现这个名字对于理解什么是 OOP 没什么帮助，也没有解释为什么我要学它用它。不用对象，我倾向于把我的 OOP 程序想象成一些演员。电子游戏可能最容易把计算机程序和演员联系在一起。例如，游戏“吃豆人”有个吃豆人演员、一些鬼魂演员、一个游戏手柄演员，以及水果演员，还有其他等。在面向对象编程的术语中，每一类演员被称为类，类的一个真正成员叫作实例。

当你用演员进行类比时，你可以把程序写成某个演员的所有行为都只跟这个演员有关。例如，吃豆人演员有在屏幕内移动的能力，所以我们可以给 `PacMan` 类加一个 `move` 方法。然后当 PacMan 需要向上移动时，我们只要调用 `pacMan.move('up')`。在吃豆人在屏幕内移动时，他的嘴不停地开闭，所以我们可以创建另一个方法，叫作 `chomp`，并在每次 PacMan 的嘴应该开闭的时候调用 `pacMan.chomp()`。在 `move` 和 `chomp` 方法写完之后，你不用担心它们是怎么工作的。你知道它们一定可以工作，但到底怎么工作的并不重要。通过这种方式，OOP 让你可以用更少的代码，以更快的速度，实现更多的功能。

类

面向对象程序有很多个对象组成，每个对象都有属性和行为。类是对一类对象的属性和行为的描述。例如，`PacMan` 类的代码包含 `move` 和 `chomp` 方法，分别描述吃豆人对象移动和咬牙动作。类只是吃豆人的描述，但并不会创建吃豆人（这是实例的工作）。使用类的最大优点在于其组织代码的方式。当你有了 PacMan 类，所有描述吃豆人的代码必须在 PacMan 类中定义。如果未来什么时候，你决定要吃豆人具有跳的能力，你很清楚要在哪里放 `jump` 方法。

继承

我们现在很清楚 PacMan 类应该长什么样子，有 move、chomp，以及（可能的）jump 方法，但游戏中的其他演员怎么办呢？最值得一提的是，鬼魂怎么办？如果你玩过“吃豆人”，你会把鬼魂当成是你的敌人。但如果你考虑如何写“吃豆人”程序，你会注意到鬼魂和吃豆人在很多方面有相似之处。在第 8 章“函数和方法”中，你学过保持代码 DRY（避免重复，Don't Repeat Yourself）的重要性，但如果我们给 PacMan 类写了 move 方法，又给 Ghost 类写了 move 方法，我们就是在制造重复。这个问题的答案是继承。

继承意味着一个类可以是另一个类的类型。举个例子，我从内华达州来，因此我是 Nevadan（内华达人）类的一员，所有 Nevadan 类的成员都有某些内华达人的专属属性和行为（比如 surviveRidiculousHeatWithLittleWater，用很少的水在极高的温度下存活），但所有 Nevadan 类的成员也是 American（美国人）类的成员。Nevadan 类（子类）集成了 American 类（父类）的所有属性和方法。所有内华达人都是美国人，但不是所有美国人都是内华达人。比如，Texan（德州人）类也继承自 American 类，但不继承自 Nevadan 类。继承的概念让代码可以在不同的类之间共享代码。PacMan 类和 Ghost 类可以继承自一个父类，我们称之为 MovableCharacter（可移动的角色）。move 方法会被定义在 MovableCharacter 类中，在 PacMan 和 Ghost 类中共享（见图 15.5）。

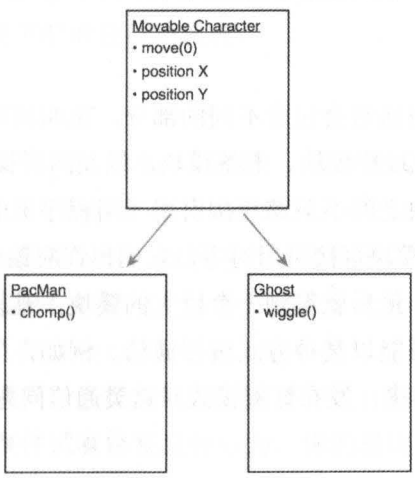

图 15.5 尽管他们各有不同，吃豆人和鬼魂共享同一个父类，MovableCharacter

实例

一个实例就像一杯可乐一样是真实的东西。PacMan 类只是一个真实的吃豆人应该是什么样子的一个概念，而 PacMan 类的一个实例是真实的吃豆人。它可以移动，可以咬牙，可以跳（也许？）——它是真实的。将类和实例分开的原因是这样你就可以创建一个类的多个实例（例如，“吃豆人”的一局游戏可以有多个鬼魂）。类和实例让你的代码整洁易懂。

设计模式

在你设计和搭建软件的过程中，记住，你的软件问题不会是独一无二的，即使你的产品是。事实上，你的问题很可能与一些已经有完善解决方案的问题非常类似。这些完善的解决方案就是所谓的设计模式。你的挑战在于选择正确的设计模式，然后正确的实现它来解决你的问题。在你遇到一个困难的编程问题时，不要重新发明轮子；使用设计模式，设计模式的意义就在于让你的生活更容易一些。

下面这个设计模式列表只是个很小的样例。这个列表的目的是给你一个概念，设计模式适合于什么场景，解决什么问题。一些设计模式描述了如何设计整个应用架构；还有些描述了如何设计应用中的一小块。你可以按照你的需求混合使用设计模式。如果你没有在这个列表中找到你的问题的答案，别担心，还有很多成熟的设计模式，它们中总会有一个能够帮你解决问题。

发布订阅

一个完整的软件应用通常会包含不同的部分，也叫模块。例如，以网络浏览器为例，它有标签模块、地址栏模块、书签模块，以及网页视图模块（见图 15.6）。要保持代码干净整洁，模块之间不应该直接引用（用程序员的话说，模块之间应该互不相识），因为如果两个模块直接引用对方，它们也许应该放在一个模块中。如果所有模块都互相直接引用，最后就得到一个巨大的模块，难以维护，几乎不可理解。不过不同的模块还是需要能以某种方式进行通信。例如，当一个标签被选中，标签关联的网页需要被显示出来。发布订阅模式对这类通信问题提供了一个解决方案。

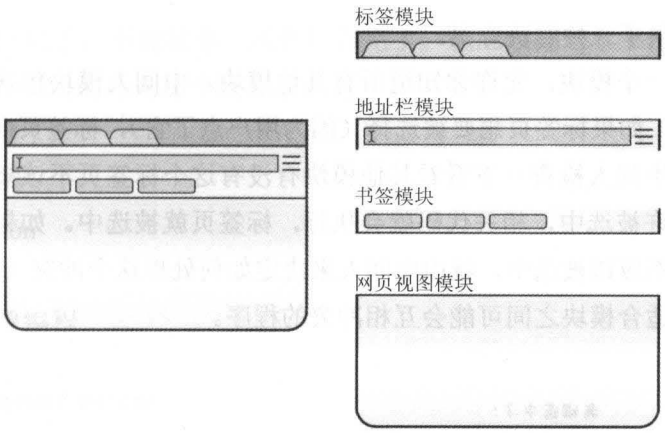

图 15.6 网络浏览器和它的模块

发布订阅模式描述了一个事件系统（和我们在第 7 章“何时使用 If、For、While”中讲的没区别）。当一个模块中发生了重要的事情，就会发布一个事件，这样其他模块就会知道这个事件发生，并做出适当的反应。一个模块必须“订阅”事件，才会在事件发生时被通知。回到我们的网络浏览器例子，当标签页被选中，标签页模块会发布一个 tabSelected 事件，地址栏模块和网页视图模块（都订阅了 tabSelected 事件）就会知道要更新它们的内容。使用事件的一个好处是，程序不管事件的来源是哪里，都会做出正确的行为。如果你要添加一个功能，允许用户通过点击一个书签模块的按钮来切换标签页，你要做的就是从书签模块发布一个 tabSelected 事件，剩下的就会自动完成。

中间人

中间人模式更像是文明版的发布订阅模式。发布订阅模式就是一大堆模块互相之间对着吼：“我是标签页！我被选择了！如果有人在意的话！”这种乱喊乱叫的方式显然在遇到冲突的时候就有问题。如果两个标签页同时发布了 tabSelected 事件，哪一个标签页会被选中？如果一个标签页发布了 tabSelected 事件，而另一个处于冻结状态（标签页在 alert 或者 prompt 窗口打开的时候会被冻结），标签页还会切换吗？应该切换吗？当模块只能通过事件进行通信，这类冲突会导致怪异行为。如果你还是只用事件试着修复这种行为，你的模块也许就会互相直接引用，直到整个发布订阅系统分崩离析。反之，你的代码需要一种调解这类冲突的方式。

中间人模式提供了一种解决方案。

中间人是一个模块，允许你知道所有其他模块。中间人模块包含用来解决冲突的代码。例如，如果标签页想要被选择（因为用户点了它），标签页向中间人请求被选中的许可。中间人检查一下看看其他模块有没有这个标签页不该被选中的理由，如果标签页允许被选中，相关代码就会执行，标签页就被选中。如果有什么理由说明这个标签页不应该被选中，就由中间人来决定如何处理这个冲突（见图 15.7）。中间人模式非常适合模块之间可能会互相冲突的程序。

图 15.7　好的中间人用对每个人都可行的方式解决冲突

单例

单例模式是一种类，只能有一个实例。每次你要创建一个单例的实例时，你其实不会得到一个新的实例，实际上你只是得到单个实例的一个引用。例如，PacMan 类可能会被实现成单例。你也许会试着在代码的不同地方创建吃豆人的实例，但实际上你只想要一个吃豆人。如果 PacMan 对象已经被创建出来，其实你不想要创建第二个吃豆人；实际上你想要的就是已经存在的吃豆人的一个引用。

总结

恭喜！你完成了，而且你甚至完成了高级主题。你应该庆祝一下。你在本书中学到的这些技能会改变你看待计算机的方式，而且也许会改变你看待世界的方式。

这些技能太有价值了，不能独享。现在你看完了，就应该把这本书借给朋友，然后你就可以一起开祖母的牙刷那个玩笑了。

本章你学到了：

- 版本控制
- Subversion
- Git
- 面向对象编程
- 类
- 实例
- 设计模式

轮到你来决定是否续写《一路编程》的下一篇章了。祝你好运！

译者介绍

佟达

毕业于哈尔滨工业大学，信息与通信工程硕士学位，现就职于ThoughtWorks，任高级咨询师。常年游走于工程与学术之间，骄傲地称自己为“会coding的科学家，懂数学的工程师”。目前专注于人工智能和深度学习领域。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

无论你是想要成为一名专业软件程序员，还是想了解如何与程序员有效沟通，或者只是对编程感到好奇，都可以从本书介绍的编程技能中受益。

来吧，让我们一起启程“编程之路”。

《一路编程》帮你打下坚实的编程基础，为实现各种编程目标做准备。作者Steven Foote自学编程，找到了可克服重重困难的最佳方式。作为一个专业Web开发者，他将带你跟随他的足迹，教给你可以用在任何现代编程语言之中的概念，不论你是面向计算机，还是智能手机、平板电脑，甚至是机器人进行编程。

- 学习如何简化及自动化诸多编程工作
- 在程序中处理不同类型的数据
- 使用正则表达式查找和处理模式
- 编写能够决定该做什么、什么时候做的程序
- 使用函数编写干净、简洁的代码
- 编写可以让其他人很容易理解和改进的程序
- 测试并调试软件，使其变得可靠
- 和程序员团队一起工作
- 学习接下来要建立一生的编程技能的步骤

读者力荐

不同于那些以语法讲解为主的编程书，本书以一种全局的视角深入浅出地讲解分析，不仅仅是编程知识，更包含了软件开发的方方面面。

——Flarum中文社区创始人 justjavac

“程序员改变世界。”这句话不假，可是很少有人真正懂编程。未来是程序员的天下。对公众而言，用清晰、易懂的语言，向所有对编程好奇、感兴趣的公众讲明编程的实用内容，是一件非常不容易的事。当我看到《一路编程》这本书时，深深地为之赞叹，原来编程的书还可以这样写啊！没有枯燥的理论说教，也没有乏味的案例，一切都是那么美好、生动。这本书，值得推荐！

——InfoQ社区编辑 刘志勇

本书让人耳目一新，看完本书，对任何类型的读者而言，都会觉得编程不再是什么高大上的事情，而是一件简单、有趣的事情，让读者有种马上就要上手编程的冲动。 本书从实践入手，让没有任何理工背景，甚至学文科的人都能很快上手编写属于他的第一个程序。

——中央财经大学信息学院副教授 海沫

近年来计算机行业可观的薪酬吸引了大批“半路出家”的外包开发人员，由于各种原因，他们在工作中往往只能以“码农”的身份从事机械性的“搬砖”类工作，缺乏对软件开发全流程和职业生涯发展的整体认知，《一路编程》为解决他们面临的这些问题提供了切实可行的指导。

——中国民生银行Java开发工程师 梅隆魁

上架建议：程序设计

ISBN 978-7-121-30478-1

9 787121 304781

定价：65.00元

Pearson
www.Pearson.com

策划编辑：符隆美
责任编辑：徐津平
封面设计：侯士卿